uOttawa

L'Université canadienne
Canada's university

FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTOCTORALES

uOttawa

L'Université canadienne
Canada's university

FACULTY OF GRADUATE AND POSDOCTORAL STUDIES

Asad B. Sayeed
AUTEUR DE LA THÈSE / AUTHOR OF THESIS

Master of Computer Science
GRADE / DEGREE

School of Information Technology and Engineering
FACULTÉ, ÉCOLE, DÉPARTEMENT / FACULTY, SCHOOL, DEPARTMENT

Developing a Minimalist Parser for Free Word Order Languages

TITRE DE LA THÈSE / TITLE OF THESIS

Stan Szpakowicz
DIRECTEUR (DIRECTRICE) DE LA THÈSE / THESIS SUPERVISOR

CO-DIRECTEUR (CO-DIRECTRICE) DE LA THÈSE / THESIS CO-SUPERVISOR

EXAMINATEURS (EXAMINATRICES) DE LA THÈSE / THESIS EXAMINERS

J. P. Corriveau

Diana Inkpen

Gary W. Slater
LE DOYEN DE LA FACULTÉ DES ÉTUDES SUPÉRIEURES ET POSTDOCTORALES /
DEAN OF THE FACULTY OF GRADUATE AND POSTDOCORAL STUDIES

Developing a Minimalist Parser for Free Word Order Languages

by

Asad B. Sayeed

A thesis submitted in conformity with the requirements
for the degree of Master of Computer Science
School of Information Technology and Engineering
University of Ottawa

Your file Votre référence
ISBN: 0-494-11403-7
Our file Notre référence
ISBN: 0-494-11403-7

Abstract

We propose a parser for free word order languages, based on ideas from the Minimalist Program. The parser simulates aspects of a human listener who necessarily begins sentence analysis before all the words have become available. We first sketch the problems that free word order languages pose. One such problem is discontinuous noun phrase constituency. Languages like Latin permit verbs, adjectives and so on to split noun phrases. We assume that the human parser assembles syntactic structures in the process of understanding a sentence; what happens to noun phrase fragments that arrive later in the derivation? Those that arrive earlier enter the existing syntactic structures, so they become less accessible. What mechanism best incorporates later fragments without undoing structures already built?

We show how difficult it is to make existing frameworks for minimalist parsing work for free word order languages and simulate realistic syntactic conditions. We briefly describe a formalism and a parsing algorithm that elegantly overcome these difficulties, and we illustrate them with detailed Latin examples. Previous formalisms for both minimalist generation and parsing tended to use cancellation of features as the primary mechanism for checking whether syntactic structures are compatible for merging into larger units. This is how words and phrases are marked as compatible and added to a larger structure. Instead, our formalism uses feature sets and unification-based operations in order to allow larger structures to acquire features from the smaller structures within them. They can then expose these features to discontinuous elements that arrive later in the derivation. In addition to the examples we provide for Latin, we provide English examples to demonstrate that this parsing algorithm can also be used with languages that require a more fixed order. After that, we discuss an implementation of this parsing algorithm written in Prolog.

We then discuss an extension to this formalism that allows it handle pro-drop languages, and we show how this can be elegantly extended to further enhance the scope of linguistic phenomena this parser can handle beyond pro-drop. Finally, we present a corpus study that justifies some of the limitations of this parser.

Acknowledgements

We would like to thank Dr. S. Szpakowicz at the University of Ottawa for his help and advice in writing this thesis. For their funding support, we would like to thank the Natural Science and Engineering Research Council of Canada. The Perseus Project[1] provided Latin texts and lexical data in a form that was easy to process automatically. In the process of working on his honours project, M. Duda assisted in the developing of the software to process the Perseus-provided data. Dr. V. Nastase of the University of Ottawa provided valuable assistance with the administrative details of submitting this thesis, and Dr. A. Weinberg of the University of Maryland at College Park helped prepare the defence. We would also like to thank Dr. B. F. McManus, emerita of the College of New Rochelle, New York, for her assistance with some of the finer points of Latin grammar.

[1] http://www.perseus.tufts.edu

Contents

Chapter 1

Introduction

The Minimalist Program as described by Chomsky [1995] and others [Uriagereka, 1998] seeks to provide an explanation for the existence of the capacity of language in humans. Syntax merits particular attention in this program, as it is syntax that mediates the interactions between the requirements of articulation and those of meaning. Minimalism characterizes syntax as the simplest possible mapping between semantics and phonology and seeks to define the terms of simplicity and determine the structures and processes required to satisfy these terms.

That the object of investigation is syntax suggests that it is possible to extract a formal and computable model of syntactic processes from minimalist investigations. Doubly so, as the concept of economy in minimalism can be said to correspond to computational complexity. There has already been a significant quantity of work done in developing a formal, computational treatment of the minimalist characterization of syntax. Stabler [1996] has developed "derivational minimalism," one way of expressing Minimalist Grammar (MG). Along with this, he has described a recognition algorithm [Stabler, 2001] for MGs in his formalism, and he and others have demonstrated that MGs are mildly context-sensitive [Michaelis, 2001], and thus believed to be sufficient to cover at least the grammars of human languages [Joshi et al., 1991, Joshi, 1985]. Niyogi [2001] has described a parser for Stabler's version of MGs that uses a form of theorem-proving to create semantic representations from a given sentence and a lexicon.

This thesis will first describe some of these approaches to MG, thereafter focusing on the deficiencies in these approaches with regard to free word order languages and incremental [1] parsing. Then it will describe one means by which attempting to realistically accomodate free word order languages in MG also leads to the development of a formal-

[1] A word we use in a slightly idiosyncratic manner that we describe in section 1.2.

ism and parsing algorithm for MG that can handle parsing in "real time," which can be particularly useful in speech applications, given the need for rapid interactions with the user. Finally it will discuss avenues in both linguistics and natural language processing that should be followed in order to get a clearer understanding of the limits of free word order phenomena and the ways in which they can by handled by deterministic parsers.

These sections are intended, as noted, to motivate the algorithm. Some readers might be interested in such aspects of these issues that would require these claims to be discussed, but that was not the point of the section: it was to explain the basic assumptions regarding why we take the approach we do. Some readers might, for example, like to see a discussion of the implications of current work in psycholinguistic experimentation or the details of possible pragmatic effects on free word order languages, but we do not extensively consider such matters here. We may discuss these issues in greater detail in future work, but they are tangential to the purpose of this work.

1.1 Free Word Order Languages

The idea of a language having the characteristic of being "free word order" is one that cannot be applied in an absolute way[2]. Word order freedom is relative; some languages have a freer word order than others. German, for instance, is freer than English in its word order in that noun phrase constituents can be permuted around the verb in second position in a non-subordinate declarative sentence. However, in most German sentences, verbs are fixed to particular positions, usually as the second or last element of a sentence. But there are languages that appear to represent extreme cases of word order freedom, such as Latin; it is claimed that in Latin, all permutations of sentences with only open category words (not prepositions, complementizers, and so on) are possible and carry the same propositional content. As this category of language is the most difficult category to explain in existing grammatical theories, we will focus on the minimalist parsing of such languages.

It matters to the parsing of sentences in free word order languages whether different orders have different meanings. It is reasonable to assume that pragmatic factors and information structure have an effect on word order[3]. The question is whether consideration of these issues is essential in the development of a parser. Is it possible to develop

[2]For instance, some people insist on referring to the phenomenon as "freer word order" [Penn and Haji-Abdolhosseini, 2003].

[3]In this instance, we say that it is "reasonable" as opposed to "necessary". This is to avoid discussion of where in the grammar these effects may appear. For instance, are they performative factors, or do they directly affect the syntax? See section 1.5.

a minimalist means of extracting the propositional content of an utterance while leaving out such matters as, for example, topic and focus?

The question can be turned around. It is possible to develop a parser that does take into account these factors? First of all, such an account requires a thorough and explicit accounting of the effects of each order. This can be particularly difficult to provide considering that some sentences with the same propositional value can have a factorial number of permutations. The second problem is the matter of context, for anything that involves pragmatic factors must somehow be integrated into the rest of the situation, both in the text and in the real world; is there a feasible way in which this information can be determined and delivered to the parser? But the third problem is potentially the most challenging: how can these features be determined in such a way that they can be used in an incremental parse (section 1.2)? This issue will be explored in the next section.

Ultimately, regardless of the pragmatics of a sentence, a parser—including the parsing mechanism presumably available within human beings—should be able to extract the propositional content of an utterance. Unless it is explicitly expressed in the sentence in an unambiguous way (and many languages give optional ways of doing this), we will assume that the parser that we are proposing herein has no awareness of context, nor does it seek to relate the sentence to a situation. This parser can be imagined as a human listener being offered an isolated sentence out of context; there is still propositional content to be extracted.

1.2 Incremental Parsing

Many recognition algorithms are supplied with the notion that as long as the task of recognizing a language is achieved, the details of recognition are of not more than passing interest. A proposed machine must be proven to specify a particular class of languages and no more, but the order in which the words of a sentence are recognized is not directly a concern, as it rarely impinges on the validity of the proof. Stabler [2001] provides just such a recognition algorithm. Niyogi [2001] extends Stabler's analysis so that it produces semantic analyses of input sentences in predicate forum, but the algorithm is still specified in terms of inference rules that do not necessarily impose an order of operations on the activities of the parser.

But the human parser does not have these luxuries. It is clear that people do not wait until the end of the sentence to perform the semantic composition of the words in the sentence. It is more likely that as soon as a word arrives, most of the semantic

connections that can be made with the previous words in the sentence *are* made—in other words, a greedy approach. A similar point is made by Kromann [1999], who argues for the use of "discontinuous trees" in an HPSG framework; greediness is actually part of his argument for the efficiency of his system.

Claims that there are processing delays need to be justified through psycholinguistic evidence. However, particularly in the case of free word order languages, psycholinguistic evidence is still not sufficiently conclusive evidence for a processing model [Sekerina, 2002] for syntax in itself, let alone a model of the temporal relationship between syntactic processing and comprehension. Consequently, the closest approximation to a fully incremental model of parsing is one that follows the relationship between syntax and semantics described above—greedily making semantic connections between words.

But why should a parsing model realistically reflect the human parser, even in the matter of incrementalism? While developing such a model is in itself a contribution to the study of the mind, there is potential for application in this as well. There are uses for syntactic processing during artificial speech processing, such as in dialogue systems; an artificial parser that operates in "real-time" as the human speech processor operates will enhance the responsiveness of speech processing systems.

The previous section alluded to implications for the parsing of free word order languages in incremental parsing model. Though different word order may have different contextual effects, it is not feasible for an incremental parser to rely on this in order to determine a correct parse or even for the parser to extract the pragmatic information. Simply put, this is due to the fact that in order to determine what path to take in extracting this information, it would be necessary for an incremental parser to have already heard several words in the sentence anyway, and from the assumption of greediness on the part of the parser, syntactic structures would already by then have been formed. It is possible that this information can be extracted at or near the end of the parse, but not in real-time[4], and thus it would not be very useful to the parse itself.

1.3 Generation versus Parsing

One of the criteria of incremental parsing is that a partial parse must always be available even though all the words necessary to produce a clear logical form are not yet available.

[4]We use realism or "real-time" almost as a synonym for incrementalism, and perhaps also locality. Incrementalism is an aspect of realism, but obviously there are further components to realism. However, in this section we are making it clear that there are many aspects of realism that we will not be dealing with. In fact, we are not even dealing with all aspects of incremental parsing, as explained in the next section.

In the process of generation, the typical assumption is that all words are available to the syntactic component when required in order to satisfy the conditions of the phonological and logical systems, to which the syntactic system is necessarily connected. The displacement of words and phrases in generation is typically considered to leave traces in their original locations, such as the detailed account of government-and-binding theory as presented by Haegeman [1994].

Parsing is often considered the inverse of generation. To parse using a particular grammatical formalism is often thus considered a search for the sequence of operations that will undo, one by one, the steps used to generate the given sentence. Some of these steps involve the movement of items from one position to another, such as the movement of agents to the subject position during passivization in English; in order to undo this movement, we would have to have the original position available[5]. During incremental parsing, this is not always the case. But we must still be able to incorporate the moved item into the structure as it arrives, given our focus on these aspects of incremental parsing. And the freer the the word order, the higher the likelihood that elements in the sentence have been displaced from positions that have not yet been reached.

Consequently, we suggest that the process of parsing, taking into account the incremental nature of the input, is not optimally characterized as merely the inverse of generation. Rather, it is the *same* process as generation (although it produces objects for the semantic system rather than the phonological system). In this case, however, rather than the words being made freely available through consultations of the lexicon when required (as in generation), the availability of the words is instead determined by their position in the phonological/orthographic input.

1.4 An Overall Design

The previous sections suggest an overall design for the parsing system whose details we will propose later in this document, in that they suggest requirements imposed on the process of parsing, such as limitations on the availability of words over time. We present an overview as Fig. 1.1.

Syntax mediates between two entities: the semantic system and the phonological/auditory system (that is responsible for input and output). The semantic system presents

[5] An example of this would be the sentence *He was suffered graciously to live.* In this sentence, it is not immediately obvious whether *He* was extracted as the object of *suffered* or as the subject of *live.* No matter one's theory, no algorithm can decide where to place the trace immediately after encountering *He* and before encountering both verbs.

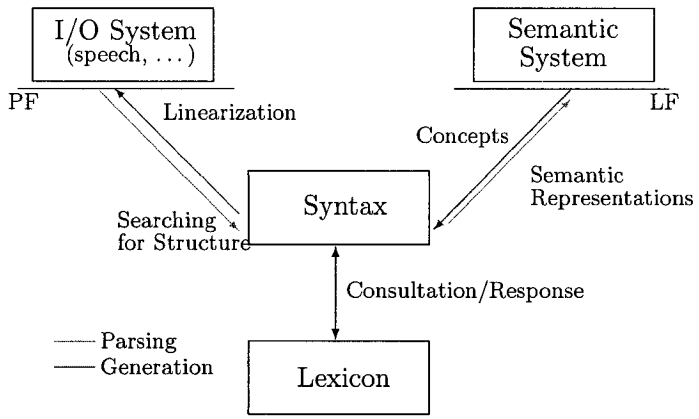

Figure 1.1: An overview of the system that contrasts generation and parsing. PF is Phonological Form, a final abstract representation of what is sent and received from the articulatory systems, be that speech, signing, or writing. LF is Logical Form, the abstract representation of meaning to be sent to the other various aspects of cognition. Conditions imposed on PF and LF in turn affect what operations are ultimately permissible in the rest of the syntactic system.

demands for the syntactic system in the form of some semantic representation; the syntactic system acquires the necessary lexical resources and assembles them into something that the phonological system can produce in an orthographic or phonetic form. The auditory system sends the words it receives one-by-one to the syntactic system, which consults the lexicon while it builds a structure from which it can achieve a semantic representation usable by the semantic system. As in section 1.2, the major "external" difference between these processes is the time at which the words are available.

Otherwise, on the face of it, the processes are very similar "internally." They both require a syntactic system that can assemble lexical objects into syntactic structures given their respective syntactic features. What differs between the two processes is their relationship to the lexical mapping. The lexicon relates phonological forms to semantic forms through syntactic features. The process of generation uses semantic forms to access lexical records in order to obtain the syntactic features required to assemble lexical items into sentences. The process of parsing receives words in the order they are heard or read, consults the lexicon to determine corresponding lexical entries, and uses the syntactic features of the words to assemble semantic structures from the meanings of words in the lexicon. In other words, generation uses semantic representations to select words for output as speech; parsing uses words to assemble semantic representations; and both of them use syntactic features to inform the process of construction of their respective output representations.

That syntactic features are used to similar ends in both processes suggests that syntactic processes do not differ greatly in an inherent way between parsing and generation, but rather they differ in the conditions on the input and on the expected form of the output. Otherwise, the structure-building processes are likely not very different for generation and parsing, as discussed in section 1.3.

1.5 Limiting the Problem

While we are proposing a parser that is "real-time" or incremental, we are not proposing a parser that is fully realistic. We are once again faced with the classic problem of separating competence from performance [Chomsky, 1965]. If we are dealing with the real-world issue of time, then why should we not deal with other real-world issues?

We are suggesting a way in which the incremental nature of the input has an effect on parsing. Since the input is provided "word-by-word", the internal processes of syntax must reflect a process that is likewise incremental. In a sense, this means that we are discussing the effect of the passage of time on the syntactic system during parsing. Over time, in real situations (that is, in the context of performance), the passage of time has many effects that impede our analysis of the system. For instance, it may be that a word is more ambiguous if we only have part of a sentence available rather than the whole sentence. Nevertheless, many interpretations of the word may not cause the parse to be successful at the end of the sentence; the selection of the correct interpretation of the word relates to the manner of lexical access and semantic representation, not the mechanics of the syntactic system. Another, more trivial example would be that of hesitation. These are issues that impede parsing when we observe its performance, but we are interested in how the system leads to a correct parse given correct information: in other words, the competence of the system—the mechanics of syntax.

We have described ways—related to performance—in which time is not a factor in explaining these mechanics. But in the previous sections, we described ways in which time does have an effect on the decisions that the syntactic system can make during parsing: the unavailability of words before the end of the parse means that the system must make decisions about relationships between words without having full information about the roles that words may play in unheard parts of the sentence. So we are compelled to consider how the system is designed to make these decisions in an environment containing insufficient information. We are thus required to consider time to be partly a matter of competence, rather than one purely set in the performative world. But we otherwise remain in the domain of abstract utterances, as we are interested solely in the

representation and processing of grammatical formalisms. Hence we avoid discussion of such things as hesitation and ambiguity as properly belonging either to performative or semantic components. We are thus able to avoid protestations of incompleteness in our discussion of parsing when those protestations stem from performative concerns that we did not consider.

Some systems appear to be so intrinsically dependent on one another that it may seem strange to some that we can discuss parsing while ignoring these connections. For instance, syntax and semantics naturally cannot be completely disconnected. But generative linguistics use various abstractions to reduce the need to discuss semantics in a syntactic context, such as θ-roles, which served as placeholders for semantic relations in syntactic accounts such as government-and-binding theory [Haegeman, 1994]. In minimalist work, particularly derivational minimalism in the style of Stabler [1996], there is little or no necessary discussion of θ-roles or semantic objects as such. This has to do with the gradual elimination of the distinction between structural and inherent case, since case is now assigned via checking[6].

For these reasons among others, it is difficult to justify any special treatment of θ (or any other semantic feature) in a syntactic discussion involving minimalist feature checking; it is a feature like any other. It made sense in a GB universe, and linguistic research is still often discussed using GB terminology. It has become easier, given minimalism, to ignore the details of semantics while working on syntactic matters that may have deep implications for the compositional semantics of sentence processing.

By the same token, we still cannot have work that is "purely syntactic" by the very definition of syntax in an incremental environment. If we assume that understanding is incremental with the arrival of words in the sentence, then clearly an order of checking implies an order of semantic absorption/composition. But one does not have to refer to the details of the semantics within this framework. This can be seen as part of what is often referred to as the "autonomy of syntax."

The work that we describe in this thesis concerns a formalism that represents a way of discussing how some languages manage to support linguistic phenomena discussed in the next chapter. It therefore may have implications for psycholinguistic research, which necessarily often involves the kinds of linguistic details to which we refer above. But this work suggests routes for psycholinguistic experimentation; it does not depend on them. Rather, we primarily arrive at this formalism through analysis of the requirements of the

[6]In GB, case was often assigned to a noun by the requirement of a semantic (θ) role provided by some other word to the position of that noun. If, as in derivational minimalism, case is assigned via an operation that combines phrases over time, then we no longer need abstract roles in certain positions to account for case itself.

syntactic phenomena that we explore in this work.

1.6 Claims and Contributions

We will now make a brief statement summarizing the above and condensing it into a statement of intent for the rest of this document. In this thesis, we demonstrate an algorithm to parse sentences containing noun phrases with discontinuous constituents, focusing on this phenomenon in Latin. We equip our formalism and algorithm with the means to handle the factors for realistic parsing that we have described in the above section 1.5.

Our algorithm handles, within a framework inspired by the Minimalist Program, most of a large category of simple declarative statements that we describe in the next chapter. We describe the formalism and algorithm first by examples, and then we proceed to a more explicit definition of the algorithm and the justifications for it. We extend the algorithm to handle Latin optional subjects, and demonstrate how this further expands the algorithm's coverage of discontinuous constituency phenomena. We then show how the formalism can be expanded to provide easily the logical forms of sentences during parsing.

After that, we suggest a means to account for the few sentences in the example sentence class that our algorithm is unable to parse by performing a study of a set of Latin prose texts in order to show the apparent infrequency of such sentences. And finally, we discuss an implementation of the algorithm that we developed in Prolog in order to verify the examples.

Given that we represent the linguistic phenomenon we describe in the next chapter (noun phrase discontinuous constituency) with an example sentence class, we have to decide what it means to build a parser that can handle these sentences: in discussing parsing, are we trying to solve an engineering problem regarding a particular language, or are trying to demonstrate principles by which this and other similar problems can be solved in general? We do not, in this work, assume in advance what is an acceptable input and then attempt to build the parsing algorithm around it: this would be an obvious course to take, but it would also be a trivial exercise. Instead, this work explores the tradeoffs between some of the external assumptions we make about the nature of the system (aspects of incrementality and locality) and the requirements of the observed syntactic phenomena. If they conflict, then we either have to evaluate whether the requirements we place on the system are too stiff or whether our analysis of the given syntactic phenomenon is not quite complete; we discuss this issue in more detail in

chapter 5.

In our view, the best way to explore such tradeoffs is to "evolve" the algorithm from the central assumptions such as incrementality, rather than from an explicit specification of the syntactic phenomena in question. To that end, we start with a more general notion of a parsing algorithm and try to pare it down so it conforms with our understanding of these phenomena. We must not reduce the algorithm to the point where our general assumptions no longer apply. The example sentences represent the syntactic phenomena that we pursue, and help clarify what is needed to build a parser informed by minimalism; we do not use examples to inspire a formal definition of what acceptable input ought to be like.

The example sentences assist in a specific way: once we have defined the formalism and parsing algorithm, we actually run the algorithm on a subset of the sentences. At each point where the algorithm as initially specified cannot handle a particular situation, we propose additional stipulations that continue to satisfy our assumptions about aspects of incrementality while minimally restricting or expanding the parser's power and complexity.

We take this approach because we feel, among other things, that it is easier on the reader to introduce the exploration by working through a couple of representative sentences and deducing what, in a minimalist world, would be required to parse them. We do assume some familiarity with GB theory and other previous approaches [Haegeman, 1994]. However, very little previous work, minimalist or otherwise, deals with aspects of the incremental parsing of discontinuous NPs, although we mention in chapter 2 some of the difficulties that an existing minimalist parsing framework might have if we were to apply it to noun phrase discontinuity.

1.7 Thesis Structure

We have organized this thesis into the following chapters:

- A chapter for the discussion of various background issues regarding the Latin language and linguistic formalisms.

- A chapter for the discussion of a formalism and parsing algorithm that handles certain difficult Latin structures while still being able to handle the equivalent English ones. This chapter is the most important one in this document and contains the bulk of what this thesis is intended to propose.

- A chapter for the discussion of an extension to the system consisting of a means to handle missing subjects in *pro-drop* languages such as Latin.

- A chapter for the discussion of a preliminary experiment in making use of Latin texts as a substitute for unavailable native speakers in demonstrating the absence of certain types of sentences predicted by the algorithms described in previous chapters.

As well, there will be a brief concluding chapter summarizing the other chapters. However, each chapter, being somewhat self-contained, will describe its own conclusions and future work. Additionally, much of the second and third chapters have been published in much-condensed form as Sayeed and Szpakowicz [2004].

Chapter 2

Background

2.1 Latin and Free Word Order

Principle-based parsing of free word order languages has been considered for a long time—see, for example, Kashket [1991]—but not much in the context of the Minimalist Program. We propose a minimalist parser, illustrated by Latin, a language that exhibits high word order freedom.

Consider for example,

pater laetus amat filium laetum
father-Nom happy-Nom loves-3Sg son-Acc happy-Acc
'The happy father loves the happy son.'

For a simple sentence such as this, in theory all $5! = 120$ permutations should be grammatical. We briefly discuss in section 3.6 whether this is really the case. This is not true of Latin sentences in which function words must be fixed in place.

It is often remarked [Pinkster, 1990] that these word orders, though semantically equivalent, differ in pragmatic import (focus, topic, emphasis, and so on). Existing elaborate accounts of the effects of pragmatic context on Latin word order [Pinkster, 1990] are rarely sufficiently formal for parser development, and do not help a parser designed to extract information from a single sentence out of context. It should still be possible to extract the propositional content without having to refer to the context; hence, we need an algorithm that will parse all 120 orders as though there were no real difference between any of them. After all, people can extract information from sentences out of context.

This amount of word order freedom is not typical of all languages considered to be free word order languages. Some languages, notable German and Japanese, allow

considerable flexibility in sentence structure at the clause level, but they never permit noun and prepositional phrase to be broken up. Latin, however, does permit this. This is typically known as discontinuous constituency. That Latin exhibits this behaviour is the primary motivation behind this work, which seeks to find a uniform way to handle Latin noun phrase discontinuities.

2.1.1 Why Latin?

Some readers may wonder why we have chosen Latin as our example language for this work. The simplest and most obvious reason is that it is a language with which we are familiar that also exhibits the property of noun phrase discontinuity. We know of many others, but we are not as familiar with them, and the overall cost of using our own knowledge of the language is much lower than that of obtaining informants for other languages.

Another important property of Latin is that the relative positions of the constituents of noun phrases do not exhibit complicated scoping effects, which may be partially the result of the absence of definite articles. Languages such as ancient Greek also exhibit similar free word order phenomena; but in the case of Greek, the presence of obligatory articles can affect the interpretation of noun phrase adjuncts depending on their position relative to the article. The semantics of noun phrases in Latin, on the other hand, are at least a little simpler.

Finally, Latin presented the additional challenge of being a language with no living native speakers. This instigated such lines of research as the corpus study we discuss close to the end of this thesis (chapter 5).

2.2 Minimalism

2.2.1 Derivational Minimalism

Stabler [1996] defines a minimalist grammar as $G = (V, Cat, Lex, F)$. V is a set of "non-syntactic features" (phonetic and semantic representations), Cat is a set of "syntactic features," Lex is the lexicon ("a set of expressions built from V and Cat"), and F is "a set of partial functions from tuples of expressions to expressions"—that is, structure-building operations such as MOVE and MERGE. MERGE, a binary operation, composes words and trees into trees. MOVE removes and reattaches subtrees; these manipulations are performed in order to *check* features. Checking ensures, among other things, that

words receive required complements. Stabler characterizes checking as the cancellation of corresponding syntactic features on the participant lexical items, but requires that features be checked in a fixed order.

Stabler [2001] proposes a minimalist recognizer that uses a CKY-like algorithm to determine membership in $L(G)$. Limiting access to features to a particular order may not work for free word order languages, where words can often appear in any order. Either duplicate lexical entires must proliferate to handle all cases, or Stabler's specification requirements of lexical entries must be relaxed. We choose the latter path.

Stabler's CKY inference rules do not themselves directly specify the order in which items are to be MERGEd and MOVEd into tree structures. This apparent nondeterminism is less significant if the feature order is fixed. Since we propose a relaxation of the ordering constraint for free word order languages, the nondeterminism will necessarily be amplified.

This dovetails with a practical goal. In free word order languages, semantically connected words can appear far apart in a sentence. In *pater amat filium laetum laetus*, *pater* and *laetus* must be merged at some point in a noun-adjective relationship. But *pater* is the subject of *amat*. In order to simulate a human listener, the parser would have to MERGE *pater* and *amat* first, despite the fact that *pater* and *laetus* form one NP; the human listener would hear and make the connection between *pater* and *amat* first.

Consequently, we propose a parser that simulates a human listener by limiting at each step in a derivation what words are accessible to ("have been heard by") the parser and by defining the precedence of the operators in a way that fully extracts the syntactic and semantic content of the known words before receiving further words. We develop a formalism to allow this while providing the flexibility needed for free word order languages, in that restrictions we place on these operators would exclude many word orders were it not for how we set up the formalism. This formalism, illustrated later by examples, reflects many of the ideas of Stabler's formalism, but it does not depend on the actual representations that Stabler uses.

2.2.2 A Note on Minimalism Itself

The reader may ask at this point, why minimalism? The Minimalist Program is a research program devoted, at least in a very large part, to determining what is absolutely necessary in the relationship between syntax and semantics. To this end, minimalist research has developed certain ways of explaining the generation of sentences by using fundamental operations that provide a guideline for simplifying older formal systems (particularly the

government-and-binding theory as described in Haegeman [1994]) of explaining syntactic phenomena.

By and large, minimalism focuses on generation. Uriagereka [1998] is a typical example; it is a large book explaining in great detail some of the foundational concepts of minimalism, but it focuses almost completely on how sentences progress from individual lexical items to a structure that can be linearized and expressed as speech. The same is true of the treatment of most linguistic phenomena in other work, including work that deals with free word order phenomena, such as Rambow and Lee [1994]. As we discussed in the introduction, parsing is not precisely the reverse of generation, although it may bear many similarities to it. Thus we must determine for the free word order phenomena we are discussing in this document what changes we need to make to the process of generation in order to produce a minimalist theory of parsing.

Free word order phenomena in the context of minimalism also need to be discussed. While such researchers as Miyagawa [1997] and Frey and Gärtner [2002] address scrambling phenomena[1], these phenomena are usually observed in languages such as German or Japanese where only high-level sentence constituents are subject to reordering. This does not tell the whole story for languages such as Latin, where it is claimed that noun phrase constituents and others can be reordered as well, can be discussed in minimalist terms. One of the few references to this kind of phenomenon is in the brief discussion in Jakielaszek [2003] of prepositional phrase discontinuity in Latin, and then yet in a generation context, used to motivate his overall point about infinitive phrases. So we have a unique opportunity to discuss a minimalist account of free word order with discontinuous noun phrases in a parsing context.

Finally, one may ask "Why not something else?" in addition to "Why minimalism?" To answer this question would be to rehash very lengthy scientific and philosophical debates about the necessity of the Minimalist Program [Uriagereka, 1998] that go beyond the scope of this document. Let it be said, instead, that the Minimalist Program offers a framework by which formal representations can be connected to a psychological reality. In this work, however, we will focus on the formal representations.

2.3 Focus on syntax

This work focuses on syntax, and one may have recognized by now that it comes from a somewhat Chomskyan perspective, and presumably one also recognizes that we are

[1]Scrambling is a name often used to refer to all phenomena wherein words in a sentence can be optionally displaced without having any effect on grammaticality or propositional content.

treating syntax as a system that effectively negotiates between several modules of the mind, including that of the lexicon (figure 1.1). But we make a major simplifying assumption: we assume that the correct lexical items are chosen for our examples and make no attempt to explain how the lexical selection occurs nor how difficulties such as ambiguity are resolved.

The immediate and obvious criticism of this approach is, however, that language is full of ambiguities, and that a system is not realistic if it cannot support such things as the late resolution of ambiguity and so on. Superficially, there is an easy response to this criticism: we are proposing some aspects of the syntactic system *given* that there is a mechanism by which the appropriate lexical items have been selected.

Going deeper, we first need to determine what it means to an exploration of syntax to include a larger lexicon with a lexical lookup and ambiguity resolution algorithm. Why consider it central to this exploration? What are the costs that it might impose? If we consider parsing to be a process of constraint-satisfaction, then we admit the chance that multiple lexical items can satisfy the constraint (that is, there is ambiguity), meaning it is necessary to try each of them at each step, eliminating the ones that fail to conform. This might vastly increase the complexity of the system.

However, *every* formalism proposed for parsing and generation using anything similar to a GB or minimalist framework makes this assumption when describing the mechanism. When demonstrating formalisms, Stabler and others usually only provide what lexical items are specifically required. One may go back later and ask what points in the algorithm must require lookups, what requirements the syntactic formalism imposes on the lookup process, and so on. But how this operates is a *separate* question from what the mechanism is doing to lexical items themselves. Only once we have formulated the syntactic mechanism can we now go back and discuss the disambiguation of indeterminate words[2].

Many issues suffer from having heavy practical import regarding language, from auditory processes to pragmatics to semantics. What are pertinent details depends on what one wants to do. For example, we are not doing work directly on word sense disambiguation (WSD). For every parser, something has to be picking out the lexical entries. If we were to discuss this in depth, we would be working on a very different thesis; those who work on WSD do not write about the parsing algorithm, either. They make some rough assumptions about distributions and work from there.

[2]A similar paradigm exists in a distantly related area of computer science research, that of programming languages, where discussion of the computation of the properties of program fragments often necessarily leads to the postulation of assumed or indeterminate variables that must be eliminated in some later, independent step [Cytron et al., 1991].

We do discuss the formal computational complexity of the system in chapter 3. And lexical matters do have an effect on overall complexity. But the reason for having a discussion of complexity at all is to demonstrate that the system we are proposing does not have such an effect on related existing frameworks (most importantly, derivational minimalism) that they become unmanageable due to our innovations alone. If we can define a mechanism whose behaviour fits within existing parameters, then we know that what we are doing is not unreasonable on the face of it.

Furthermore, any substantial claim about the additional complexity introduced by lexical matters must assume many things about the structure of the lexicon: for instance, the way in which similar items in the lexicon are stored. This is precisely one of the reasons why this is a separate issue: if one stores the items efficiently, then the problem of the additional complexity of parsing caused by lexical search does not exist. Efficient storage of the lexicon is a separate matter from the mechanism used to compose lexical items into syntactic structures.

Before we even begin to think of how our lexicon is actually structured, we need to determine what constraints language actually imposes on it. To determine these constraints, we have to work with linguistic (in this case, syntactic) phenomena. One such phenomenon is discontinuous constituency, to the parsing of which in Latin we devote the rest of this thesis.

Chapter 3

The Parser

3.1 Lexicalized Grammatical Formalism

We briefly describe enough of the lexicon structure to assist in understanding the parsing examples in section 3.3 and the description of the algorithm in section 3.4. A lexical entry has the form

$$\alpha : \Gamma$$

α is a word's phonetic or orthographic representation as required. Γ is a set of feature structures, henceforth called "feature sets."[1] Γ contains feature paths, described below. But more fundamental are the feature bundles from which feature paths are constructed.

Feature bundles are required given the fact that highly inflected languages often compress several features into a single morpheme. Feature bundles provide the means to check them simultaneously. A feature bundle is represented as follows:

$$\beta(\tau_1 : \phi_1, \ldots, \tau_n : \phi_n) : \delta, n \geq 1$$

β is the feature checking status. It can be unchecked (unch), UNCHECKED (UNCH), checked (ch), CHECKED (CH), unchecked-adjoin (unch+), checked-adjoin (ch+). When feature bundles are checked against one another, β can change. The algorithm will define and the examples will illustrate the relation between feature checking status symbols during

[1] There is a third, hidden entity here: the semantic representation of α. We leave it implied that parsing operations perform semantic composition; the formal specification of this is left for future work, but it can be specified in the lambda calculus as in Niyogi [2001] or via theta roles, and so on. Nevertheless, this work is ultimately directed towards laying the syntactic groundwork for the extraction of semantic content from free word order sentences.

the checking.

Each τ is a feature type. It can be one of such things as case, gender, number, and so on. Each ϕ is a feature value such as Nom (nominative), Sg (singular), and so on. The correspondence between feature types and features is required for the unification aspect of the checking operation, again demonstrated in the examples.

δ is direction. An item ι carrying the feature bundle can only check the bundle with a bundle on another item to the δ of ι. δ can be *left* or *right*, and δ is omitted when the direction does not matter (a frequent situation in free word order languages).

A feature path has the form:

$$\pi \rightarrow \Gamma$$

π is a feature bundle and Γ is a feature set. A feature path makes Γ inaccessible for checking until π has been checked. Γ can be empty, in which case the $\rightarrow$ is not written.

A feature set is simply an unordered list of unique feature paths: $\{\eta, \ldots\}$. These sets allow each η to be checked independently of the others. In our representation of lexicon entries, we leave out the braces if there is only one η.

3.2 Data Structures and Operations

A parse (also known as a derivation) consists of a number of steps. Each step is of the form:

$$\Phi \mid \Psi$$

Ψ is the queue of incoming words. It is used to simulate the speaker of a sentence. The removal of words is restricted to the left end of the queue. A word is shifted onto the right end of Φ, given conditions described below. Shifting is equivalent to "hearing" or "reading" the next word of a sentence. Φ is a list, the processing buffer. It is a list of trees whereon the parser's operations are performed. The initial state of a parse has an empty Φ, and the final successful state has an empty Ψ and a single tree in Φ without any unchecked features. A parse fails when Ψ is empty, Φ has multiple trees or unchecked feature bundles, and no operations are possible.

Tree nodes are like lexicon entries, maybe with some features checked. The form is $[\alpha\ \Gamma]$ if it is the node with word α closest to the root, or α if it is a lower node. A single node is also a tree.

At each step, the parser can perform one of three operations: MOVE a node on a tree to a higher position on the same tree, MERGE two adjacent trees, or shift a word from the input queue to the processing buffer in the form of a node with the corresponding features from the lexicon.

MERGE and MOVE are well known in the minimalist literature [Uriagereka, 1998], but for parsing we present them a little differently. Their operation is defined in section 3.4 and illustrated in the subsequent examples. In this parser, MERGE finds the first[2] two compatible trees from the processing buffer with roots α and β, and replaces them with a tree with either α or β (see section 3.4 for how one of them is chosen) as the root and the original trees as subtrees. MOVE finds a position α in a tree (including the possible future sister of the root node) that commands[3] a compatible subtree; the subtree is replaced by a trace at its original position and merged with the item at its new position. Adjunct movement and specifier movement is possible. Our examples only show adjunct movement. MOVE acts on the first tree for which this condition exists.

There are locality conditions for MERGE and MOVE. For MERGE, the trees must be adjacent in the processing buffer. For MOVE, the targeted tree positions must be the ones as close as possible to the root.

Compatibility is determined by feature checking. It relies on unification and unifiability tests. Sometimes checking succeeds without changing the checking status of its participants, because further checking may be required. These aspects of checking are also described in the algorithm definition and in the examples.

At every step, MOVE is always considered before MERGE. This is to minimize the number of feature-compatible pairs of movement candidates within the trees in the processing buffer. If no movement is available in any tree, the parser looks for adjacent candidates to MERGE, starting from left to right, in order to make use of the semantic material that arrived with the earlier words. If neither MERGE nor MOVE is possible, a new word is shifted from the input queue.

3.3 Examples of Parsing

For the sake of readability, we present the examples prior to the formal definition of the algorithm; we use the examples to illustrate some of the motivation for our definition of

[2]We scan from the left (the beginning of the input), since we want to form semantic connections as soon as possible after a word has entered the system; the leftmost words have been there for the longest time. However, since the trees closer to the left end of Φ tend to have their features already checked, processing usually affects recently shifted items more.

[3]Node ξ *commands* node ζ if ξ's sister dominates (is an ancestor of) ζ.

the algorithm in section 3.4.

The Latin examples are selected for a specific purpose: they are examples of discontinuous constituency, and their existence is there to motivate what is really an exploration of what discontinuous constituency requires in a context of minimalism and incrementalism. All that is required are representative examples of discontinuous constituency, since there are an infinite number of possible discontinuous examples, but they all follow a particular pattern: the NP is interrupted by some other non-adjunct, non-constituent phrase. Examples help to explain our design decisions

The English examples are there to demonstrate a simple but important point: that the algorithm continues to parse correctly equivalent representative sentences in a language that does not have those characteristics but may have others. The examples have the salutary effect of allowing for more clarifications of what the algorithm needs to contain, but the fundamental points are made by the Latin examples.

We rely heavily on the examples in the discussion of how and why we define our algorithm the way we do. This also has some important and even necessary consequences for a general solution. We provide a more formal description of the overall algorithm, but some of the checking details are best described in terms of reactions to the dilemmas occasionally reached at certain points in the process of parsing. The details of the checking process, by their very nature, can only be determined by exploring the linguistic phenomena themselves. When dealing with a complex linguistic phenomenon, one has to start with slices of it, and adapt the system from there, describing the motivation for some decisions based on the phenomena one observes.

To restate the above, our discussion by examples discovers systematically, by going through derivations in detail, what the pitfalls and requirements of the incremental parsing of discontinuous constituency are. So there are going to be aspects of it that look like stipulations specific to particular situations, even though they follow from the basic assumptions and the linguistics data, and thus really are not as stipulative as they seem.

3.3.1 Latin Examples

Here is the initial lexicon:

pater: unch(case:Nom, num:Sg, gnd:Masc)
filium: unch(case:Acc, num:Sg, gnd:Masc)
amat: {UNCH(case:Nom, num:Sg), UNCH(case:Acc)}
laetus: unch+(case:Nom, num:Sg, gnd:Masc)

laetum: unch+(case:Acc, num:Sg, gnd:Masc)

Inflection in Latin is often ambiguous. More entries for *laetum* would exist in a realistic lexicon. As disambiguation is not in the scope of this thesis, we assume for simplicity that the only entries in the lexicon are those useful for our examples.

Example #1: *pater laetus filium amat laetum*

This example illustrates the basic machinery of the parser, but it also demonstrates how the parser handles the discontinuous constituency of phrases, in this case noun phrases. *filium laetum* ("the happy son") is split across the verb. We begin with an empty processing buffer:

| pater laetus filium amat laetum

pater and *laetus* are shifted into the buffer one by one. They are "heard"; the lexicon is consulted and the relevant features are attached to them. (For the sake of brevity, we will combine multiple steps, particularly when a word is heard and some merge happens immediately as a result.)

[pater unch(case:Nom, num:Sg, gnd:Masc)]
[laetus unch+(case:Nom, num:Sg, gnd:Masc)]
| filium amat laetum

laetus adjoins to *pater*. There is no unifiability conflict between the features of both words. If a conflict had occurred, there would be no MERGE. The lack of conflict here indicates that the adjunction is valid. Thus, the feature on the adjective is marked as checked. Since adjunction is optional, and further adjunctions are theoretically possible, the feature on the noun is not checked yet.

([pater unch(case:Nom, num:Sg, gnd:Masc)]
  pater
  [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
| filium amat laetum

We shift *filium* into the buffer. *filium* cannot be absorbed by the *pater* tree. So we shift *amat*.

([pater unch(case:Nom, num:Sg, gnd:Masc)]
  pater

[laetus ch+(case:Nom, num:Sg, gnd:Masc)])
[filium unch(case:Acc, num:Sg, gnd:Masc)]
[amat {UNCH(case:Nom, num:Sg), UNCH(case:Acc)}]
| laetum

Can anything be merged? Yes, *filium* checks a feature bundle on *amat* that was the one looking for another compatible bundle in order to project to the root of the new tree. When *filium*'s feature bundle checks with the corresponding bundle on *amat*, several things occur:

1. *amat*'s feature bundle's status changes from UNCH to CH, and *filium*'s bundle's status changes from unch to ch.

2. *amat* projects: a new tree is formed with *amat* and its features at the root. This is specified in the feature bundle: the capitalized form indicates that *amat* is looking for a constituent to fill one of its semantic roles.

3. *amat*'s feature bundle is unified with that of *filium* and replaced with the unification result; in other words, it acquires *filium*'s gender. *filium* does not gain any features, as its bundle is not replaced with the unification result. Only the projecting item *amat* is altered, as it now dominates a tree containing *filium* as a constituent. The non-projecting item becomes a subtree, inaccessible for MERGE at the root and thus not needing to reflect anything about *amat*.

Table 3.1 describes the interactions between feature bundle checking status types. In all four cases, only the item containing bundle 2 projects and forms the root of the new tree. A unifiability test occurs in each checking operation, but replacement with the unification result happens only in the feature checked on the projecting item (bundle 2) in the first case. No combinations other than these are valid for checking. The relation only allows us to check unch with UNCH feature bundles, since UNCH bundles indicate that their bearers project; there must be exactly one projecting object in any MERGE or MOVE. unch+ check with CH feature bundles, because their CH (and feature-compatible) status will have resulted from a merge with an item that can accept the adjunct in question. unch+ check with any compatible unch or ch feature bundles, as this indicates that the target of adjunction has been reached. We consider this analysis of checking relations to be exhaustive, but we save the rigorous elimination of other combinations for future work.

The result of the MERGE of *filium* and *amat*:

Bundle 1	Bundle 2	after checking:	Bundle 1	Bundle 2	Replace bundle 2 with unif. result?
unch	UNCH		ch	CH	Y
unch+	CH		unch+	CH	N
unch+	unch		ch+	unch	N
unch+	ch		ch+	ch	N

Table 3.1: Checking status interactions.

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([amat {UNCH(case:Nom, num:Sg),
          CH(case:Acc, num:Sg, gnd:Masc)}]
    [filium ch(case:Acc, num:Sg, gnd:Masc)]
    amat)
| laetum
```

MERGE occurs at the roots of trees. It treats each tree as an encapsulated object and does not *search* for places within trees to put objects. In more complicated sentences, searching would require more involved criteria to determine whether a particular attachment is valid. For the sake of minimality, we have developed a process that does not require such criteria, but only local interaction at a surface level. In doing so, we preserve our locality requirements.

The only attachments to trees that are valid are those that are advertised at the root. MOVE within trees takes care of remaining checkable features given criteria of minimality described above.

Now, *pater* is adjacent to *amat* and can thus MERGE with it, checking the appropriate bundle.

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
          CH(case:Acc, num:Sg, gnd:Masc)}]
    ([pater ch(case:Nom, num:Sg, gnd:Masc)]
        pater
        [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
    (amat
        [filium ch(case:Acc, num:Sg, gnd:Masc)]
        amat) )
| laetum
```

In the processing buffer, no more features can be checked. The system needs to process *laetum*. It moves in and presents a problem:

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
          CH(case:Acc, num:Sg, gnd:Masc)}]
    ([pater ch(case:Nom, num:Sg, gnd:Masc)]
        pater
        [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
    (amat
        [filium ch(case:Acc, num:Sg, gnd:Masc)]
        amat) )
[laetum unch+(case:Acc, num:Sg, gnd:Masc)] |
```

To what can *laetum* attach? We have defined the merge operation so that it cannot search inside a tree—it must operate on objects in the buffer. Fortunately, the rules we have defined allow an item with an adjunct feature bundle to be merged with another item with a ch projecting feature bundle if both bundles can be correctly unified. The adjunct feature remains unch until movement causes a non-projecting non-adjunct feature to be checked with it.

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
          CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        ([pater ch(case:Nom, num:Sg, gnd:Masc)]
            pater
            [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
        (amat
            [filium ch(case:Acc, num:Sg, gnd:Masc)]
            amat) )
    [laetum unch+(case:Acc, num:Sg, gnd:Masc)]) |
```

The rules for movement seek out the highest two positions on the tree that can be checked with one another. *filium* and *laetum* are precisely that. *filium* is copied, checked, and merged with *laetum*. For the sake of convention, we mark the original position of *filium* with a trace (<*filium*>).

```
([amat {CH(case:Nom, num:Sg, gnd:Masc),
          CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
```

```
([pater ch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
(amat
    <filium>
    amat) )
([filium ch(case:Acc, num:Sg, gnd:Masc)]
    filium
    [laetum ch+(case:Acc, num:Sg, gnd:Masc)]) ) |
```

filium dominates because *laetum* is only an adjunct. All features are now checked, and the parse is complete.

Example #2: *pater laetus a filio laeto amatur*

This sentence is in the passive voice. Passives in Latin are very similar to passives in English. *a filio laeto* is similar to an agent *by*-phrase. This requires new lexicon entries for all the words except for *pater* and *laetus*. The additional entries:

```
a: UNCH(case:Abl):right → unch(by:0)
filio: unch(case:Abl, num:Sg, gnd:Masc)
laeto: unch+(case:Abl, num:Sg, gnd:Masc)
amatur: {UNCH(case:Nom, num:Sg, gnd:Masc), UNCH(by:0)}
```

The *by*-feature is similar to Niyogi's 2001 solution for *by*-phrases in English. 0 is there as a place-holder, since the *by*-feature does not have multiple values; it is either present or absent. We also use 0 to indicate that the feature *must* be present in order for the feature bundle to be unifiable with a corresponding UNCH feature bundle. *a* is a preposition in Latin with other uses (like *by* in English); making it necessarily attach to a verb as the deliverer of an agent requires a special feature.

The *by*-feature is the second element along a feature path. Before *a* can be merged with a verb, it must first be merged with a complement in the ablative case. This complement is directionally specified (to the right of *a*).

Let us begin:

```
| pater laetus a filio laeto amatur
```

pater and *laetus* are each shifted to the processing buffer and adjoin:

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
| a filio laeto amatur
```

a and *filio* enter; *filio* is a noun in the ablative case and can be checked against *a*.

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([a CH(case:Abl, num:Sg, gnd:Masc):right → unch(by:0)]
    a
    [filio ch(case:Abl, num:Sg, gnd:Masc)])
| laeto amatur
```

filio checks the case feature on *a* immediately. There will be an adjective that needs to adjoin to *filio*, but it has not yet been heard; meanwhile, the system has a preposition and a noun immediately ready to work with. The mechanism of unification allows *a* to advertise the requirements of *filio* for future adjunction. There is no reason for the system to wait for an adjective, as it obviously cannot know about it until it has been heard. The noun does not need an adjective and only permits one if one is available.

As before, *laeto* is absorbed and attached to *a*.

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([a CH(case:Abl, num:Sg, gnd:Masc):right → unch(by:0)]
    (a
        a
        [filio ch(case:Abl, num:Sg, gnd:Masc)])
    [laeto unch+(case:Abl, num:Sg, gnd:Masc)])
| amatur
```

This adjunction occurs because unification and replacement have caused *a* to carry the advertisement for a Sg, Masc adjunct in its now-CH feature that previously only specified ablative case.

MOVE connects *filio* and *laeto*, leaving <*filio*>:

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
```

```
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([a CH(case:Abl, num:Sg, gnd:Masc):right → unch(by:0)]
    (a
        a
        <filio>)
    ([filio ch(case:Abl, gnd:Masc, num:Sg)]
        filio
        [laeto ch+(case:Abl, num:Sg, gnd:Masc)]) )
| amatur
```

The buffer now contains two trees. Neither of the roots of these trees have any features that can be checked against one another. *amatur* is heard:

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([a CH(case:Abl, num:Sg, gnd:Masc):right → unch(by:0)]
    (a
        a
        <filio>)
    ([filio ch(case:Abl, gnd:Masc, num:Sg)]
        filio
        [laeto ch+(case:Abl, num:Sg, gnd:Masc)]) )
[amatur {UNCH(case:Nom, num:Sg, gnd:Masc), UNCH(by:0)}] |
```

The case feature was checked on *a*, so the *by*-feature is available for checking with the verb. Recall that a MERGE in the processing buffer only occurs between adjacent elements. In the next step *amatur* can only merge with *a*.

```
([pater unch(case:Nom, num:Sg, gnd:Masc)]
    pater
    [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
([amatur {UNCH(case:Nom, num:Sg, gnd:Masc),
          CH(by:0, case:Abl, num:Sg, gnd:Masc)}]
    ([a CH(case:Abl, num:Sg, gnd:Masc):right → ch(by:0)]
        (a
            a
            <filio>)
        ([filio ch(case:Abl, gnd:Masc, num:Sg)]
```

```
        filio
        [laeto ch+(case:Abl, num:Sg, gnd:Masc)]) )
amatur) |
```

Note that the *by*-feature on *amatur* has been unified with all the features on the feature path of *a*. This is necessary in case the order had been *pater laetus amatur a filio laeto*. In that situation, *a filio* would have been merged with *amatur* before *laeto* would be heard, since *laeto* is an adjunct. This mechanism ensures that permission for the attachment of an adjunct to *filio* is exposed at the root of the tree dominated by *amatur*. Here it is not an issue, but the operator covers this possibility.

Merging with *pater* is the next and final step:

```
([amatur {CH(case:Nom, num:Sg, gnd:Masc),
          CH(by:0, case:Abl, num:Sg, gnd:Masc)}]
    ([pater ch(case:Nom, num:Sg, gnd:Masc)]
        pater
        [laetus ch+(case:Nom, num:Sg, gnd:Masc)])
    (amatur
        ([a CH(case:Abl, num:Sg, gnd:Masc):right → ch(by:0)]
            (a
                a
                <filio>)
            ([filio ch(case:Abl, gnd:Masc, num:Sg)]
                filio
                [laeto ch+(case:Abl, num:Sg, gnd:Masc)]) )
        amatur) ) |
```

All the unch features have been exhausted, and the parse is complete.

3.3.2 English examples

We give a couple of English examples (active and passive) to demonstrate that our approach continues to work well for more fixed-order languages. Our example: *The happy father loves the happy son.*

Here is the initial lexicon:

```
the: UNCH(type:N):right → unch(type:N)
happy: unch+(type:N):right
father: unch(type:N, num:Sg)
```

```
son: unch(type:N, num:Sg)
loves: {UNCH(type:N, num:Sg):left, UNCH(type:N):right}
```

English not having explicit case for most nouns requires that the type be used (N for nouns) instead to distinguish directly between categories; in the Latin examples, it was possible not to explicitly encode for noun phrases since the case, gender, number, and checking type sufficiently distinguished between categories.

Example #1: *the happy father loves the happy son*

Now we begin with the sentence:

```
| the happy father loves the happy son
```

the and *happy* are shifted first, but nothing can happen until *father* is shifted:

```
[the UNCH(type:N):right → unch(type:N)]
[happy unch+(type:N):right]
[father unch(type:N, num:Sg)]
| loves the happy son
```

happy and *father* must MERGE. They are the first compatible pair in the processing buffer.

```
[the UNCH(type:N):right → unch(type:N)]
([father unch(type:N, num:Sg)]
    [happy ch+(type:N):right]
    father)
| loves the happy son
```

Now *father* becomes accessible to *the*.

```
([the CH(type:N, num:Sg):right → unch(type:N)]
    the
    ([father ch(type:N, num:Sg)]
        [happy ch+(type:N):right]
        father))
| loves the happy son
```

the continues to make an N feature available due to the feature path, allowing the phrase to continue to be an NP. We shift *loves*.

```
([the CH(type:N, num:Sg):right → unch(type:N)]
    the
    ([father ch(type:N, num:Sg)]
        [happy ch+(type:N):right]
        father))
[loves {UNCH(type:N, num:Sg):left, UNCH(type:N):right}]
| the happy son
```

Note that both *father* and *the* both have N features that would be compatible with *loves*. But since all MERGEs happen before shifting, only the entire NP including *the* is available. MERGE checkes the N feature on *the*. (This causes the unch feature to be unified with the UNCH feature, causing them both to acquire the value Sg.)

```
([loves {CH(type:N, num:Sg):left, UNCH(type:N):right}]
    ([the CH(type:N, num:Sg):right → ch(type:N, num:Sg)]
        the
        ([father ch(type:N, num:Sg)]
            [happy ch+(type:N):right]
            father))
    loves)
| the happy son
```

It should be remembered from the Latin examples that no features on feature paths being checked against one another may contradict. This enforces the requirement on *loves* for a singular noun phrase.

the happy son are shifted into the buffer and merged into a tree exactly like that of *the happy father*.

```
([loves {CH(type:N, num:Sg):left, UNCH(type:N):right}]
    ([the CH(type:N, num:Sg):right → ch(type:N, num:Sg)]
        the
        ([father ch(type:N, num:Sg)]
            [happy ch+(type:N):right]
            father))
    loves)
([the CH(type:N, num:Sg):right → unch(type:N)]
    the
    ([son ch(type:N, num:Sg)]
        [happy ch+(type:N):right]
        son)) |
```

Then finally *the* is merged with *loves*.

```
([loves {CH(type:N, num:Sg):left, CH(type:N, num:Sg):right}]
    (loves
        ([the CH(type:N, num:Sg):right → ch(type:N, num:Sg)]
            the
            ([father ch(type:N, num:Sg)]
                [happy ch+(type:N):right]
                father))
        loves)
    ([the CH(type:N, num:Sg):right → ch(type:N)]
        the
        ([son ch(type:N, num:Sg)]
            [happy ch+(type:N):right]
            son))) |
```

All features are checked and the parse is complete. Note that MOVE was never required, since every feature bundle could be checked with a feature bundle on a neighbouring word.

Example #2: *the son is loved by the father*

For this passive example, we have dropped the adjective *happy*, as its MERGE into the noun phrase was amply demonstrated by the preceding English and Latin examples. In any case, this passive example is more complicated than the Latin passive example due to the presence of the auxiliary *is*. We propose the following additional lexicon entries:

```
is: {UNCH(type:N, num:Sg):left, UNCH(type:V):right}
loved: UNCH(by:0):right → unch(type:V)
by: UNCH(type:N):right → unch(by:0)
```

So, to begin:

```
| the son is loved by the father
```

the son move into the buffer and form a tree:

```
([the CH(type:N, num:Sg) → unch(type:N)]
    the
    [son ch(type:N, numSg)])
| is loved by the father
```

is loved by cannot be merged, since they serially depend on each other. However, *the son* can be merged with *is*:

```
([is {CH(type:N, num:Sg):left, UNCH(type:V):right}
    ([the CH(type:N, num:Sg) → ch(type:N, num:Sg)]
        the
        [son ch(type:N, numSg)])
    is)
[loved UNCH(by:0):right → unch(type:V)]
[by UNCH(type:N):right → unch(by:0)]
| the father
```

the father shift and become a tree.

```
([is {CH(type:N, num:Sg):left, UNCH(type:V):right}
    ([the CH(type:N, num:Sg) → ch(type:N, num:Sg)]
        the
        [son ch(type:N, num:Sg)])
    is)
[loved UNCH(by:0):right → unch(type:V)]
[by UNCH(type:N):right → unch(by:0)]
([the CH(type:N, num:Sg) → unch(type:N)]
    the
    [father ch(type:N, num:Sg)]) |
```

by and *the father* can now merge.

```
([is {CH(type:N, num:Sg):left, UNCH(type:V):right}
    ([the CH(type:N, num:Sg) → ch(type:N, num:Sg)]
        the
        [son ch(type:N, num:Sg)])
    is)
[loved UNCH(by:0):right → unch(type:V)]
([by CH(type:N, num:Sg):right → unch(by:0)]
    by
    ([the CH(type:N, num:Sg) → ch(type:N)]
        the
        [father ch(type:N, num:Sg)])) |
```

The chain reaction continues. The checking of the N feature on *by* make the by-feature available to *loved*.

```
([is {CH(type:N, num:Sg):left, UNCH(type:V):right}
    ([the CH(type:N, num:Sg) → ch(type:N, num:Sg)]
        the
        [son ch(type:N, num:Sg)])
    is)
([loved CH(by:0, type:N, num:Sg):right → unch(type:V)]
    loved
    ([by CH(type:N, num:Sg):right → ch(by:0)]
        by
        ([the CH(type:N, num:Sg) → ch(type:N)]
            the
            [father ch(type:N, num:Sg)]))) |
```

Note that we have a problem here: *loved* acquires a type feature through checking with *by*, and this causes a contradiction in its feature path. This is not a contradiction between two feature paths; it is a contradiction caused by a valid merge within a *single* feature path. The later feature bundle's requirements, however, should not be compromised by a previous merge. Therefore, on a feature path, we allow future feature bundles to override previous feature bundles, as the contradiction occurred earlier in the derivation and thus has the potential to distort future requirements. In our example, that means that the only features on the path available for unification in the next step are the V and Sg features. For the sake of consistency, we also assume that the absence of the singleton by-feature from the next feature bundle also indicates that it is overridden: *loves* should not be restricted to accumulating passives. The same process of overriding singletons must also apply to adjuncts even for singletons in the last unchecked feature on a path for precisely the same reason—singletons like the by-feature should not be able to prevent unanticipated non-complements.

Finally, *loved by the father* can be merged with *the son is.*

```
([is {CH(type:N, num:Sg):left, CH(type:V, num:Sg):right}
    is
        ([the CH(type:N, num:Sg) → ch(type:N, num:Sg)]
            the
            [son ch(type:N, num:Sg)])
        is)
    ([loved CH(by:0, type:N, num:Sg):right → ch(type:V)]
        loved
        ([by CH(type:N, num:Sg):right → ch(by:0)]
```

```
by
([the CH(type:N, num:Sg) → ch(type:N)]
    the
    [father ch(type:N, num:Sg)])))) |
```

All features are checked, and the parse is complete.

We discuss an implementation of this parser in chapter 6; in appendix B, we provide sample output that demonstrates that these two Latin and two English sentences can be parsed using our implementation to produce the structures that we demonstrate in this chapter.

3.4 The Algorithm

3.4.1 Preliminaries

We only need to assume the existence of two fundamental operations, both closely related: UNIFY and UNIFIABLE?. The former unifies the flat feature structures contained in feature bundles; the latter checks whether they are compatible and returns a boolean value. Given the bundles

$$\beta_1(\rho_1 : \chi_1, \ldots, \rho_n : \chi_n) : \delta_1$$

and

$$\beta_2(\tau_1 : \phi_1, \ldots, \tau_n : \phi_n) : \delta_2$$

we say that they are unifiable (UNIFIABLE? returns true) if there is no $\rho = \tau$ where the corresponding $\chi \neq \phi$.

If the two bundles are UNIFIABLE?, then we can UNIFY their features. This is simply the union of the sets of their two features.

Usually after a successful unifiability test or a unification, feature-checking occurs. This is a basic feature of both MERGE and MOVE, and is illustrated in detail in previous sections; MERGE and MOVE are also illustrated in fair detail: MERGE produces from two trees a new tree containing the original two and dominated by a copy of the root node of one of them, and MOVE detaches a subtree from a tree, leaving a trace, and performs almost the same act as MERGE with a higher subtree. What is not described explicitly is the algorithm by which operations are selected. Consequently, at this point we provide such a description of the algorithm, and in the process of doing this we will also provide an explicit definition of MERGE and MOVE themselves.

MERGE takes two nodes t_1 and t_2:

1. For each feature bundle b_1 in t_1

   (a) For each feature bundle b_2 in t_2

      i. If the features in b_1 and b_2 are unifiable (all the subsequent cases should be assumed to be symmetrical in that, for example, it should not matter whether b_1 is unch and b_2 is UNCH or vice versa—the node containing the UNCH bundle will always become the root)

         A. If b_1 is unch and b_2 is UNCH

            - Set b_1 to ch and b_2 to CH.
            - Set the features of b_1 and b_2 to the result of UNIFY on the two bundles.
            - Copy t_2 as t_3 and make t_1 and t_2 children of t_3.
            - Return success.

         B. Else if b_1 is unch+ and b_2 is unch or ch

            - Set b_1 to ch+.
            - Copy t_2 as t_3 and make t_1 and t_2 children of t_3.
            - Return success.

         C. Else if b_1 is unch+ and b_2 is CH

            - Copy t_2 as t_3 and make t_1 and t_2 children of t_3.
            - Return success.

         D. Otherwise, return failure.

      ii. Otherwise, return failure.

Note that cases B and C differ in that in B an adjunct feature bundle (b_1) is being checked with a bundle on the node to which the adjunct will be finally adjoined. C, on the other hand, represents the case where b_1 is being checked with a bundle on a node that is an ancestor of the adjunct's proper counterpart; in this case, the CH for b_2 indicates that the node has already been checked and unified with a compatible ch feature bundle.

MOVE assumes that we can access t_1 and t_2's parents. It also assumes that t_2 is the lower tree (meaning that t_1 either commands t_2 or dominates t_2).

1. Call MERGE on t_1 and t_2. The result is t_3.
2. If MERGE fails, fail.
3. Detach t_2 from its parent, replacing it with a trace.

4. Detach t_1 from its parent t_4.

5. Attach t_3 as a child of t_4.

So MOVE is dependent on MERGE, given that they depend on the same checking relations. However, on a pure MERGE of two trees in the processing buffer, the unch+ with ch combination would never occur, seeing as ch features only exist after a subtree has been absorbed into a tree through a previous MERGE.

3.4.2 PARSE and OP-SEARCH

PARSE is the overall process of parsing. It is incremental and recursive. It makes use of OP-SEARCH, which finds the best operation to perform on the tree and performs it. It takes two arguments, an initially empty processing buffer and the input queue, provided as the tuple *(proc, input)*.

1. Run OP-SEARCH on *proc*.

2. If OP-SEARCH fails (no operation available)

   (a) If *input* is non-empty, shift a word from *input* and call PARSE again.

   (b) If *input* is empty

      i. If there is more than one tree in *proc*, return failure.

      ii. If there is only one tree in *proc*, visit every node in the tree.

         A. If there is an unchecked feature in any node, return failure.

         B. Else, return success.

3. Call PARSE again.

The parsing process is rather simple, except for the case where no operation is possible. Then either a new word must be shifted, or there are no new words. If there are no new words, then there should be no unchecked features—otherwise, if unchecked features trigger no new operations, the parse has failed. There should also not be multiple trees for the same reason; the parse will never complete. The parse succeeds when there are no new operations, and there is a single tree with no unchecked features. Beyond that, as long as there is an operation, the process keeps iterating.

This is OP-SEARCH:

1. For each tree in *proc*

   (a) Call MOVE-SEARCH.

   (b) If it succeeds, return success.

2. Call MERGE-SEARCH on *proc*.

3. If it succeeds, return success.

4. Otherwise, return null.

3.4.3 MERGE-SEARCH and MOVE-SEARCH

We have described the elementary operations of unification and the operation of MERGE and MOVE themselves, and we have described the overall incremental parsing algorithm that directs the system. However, this parsing algorithm depends on searching *proc* for potential MOVEs and MERGEs. So here we define two processes MOVE-SEARCH and MERGE-SEARCH. MERGE-SEARCH is simpler (in that it only requires searching a list of root nodes rather than entire trees), so we begin with it.

MERGE-SEARCH assumes we can know the length of *proc*, and it accesses *proc* as an array indexed from 0. Thus,

1. For i from 0 to the length of *proc* minus 1

   (a) If the MERGE of $proc_i$ and $proc_{i+1}$ succeeds

      i. Insert the result after $proc_{i+1}$.

      ii. Remove $proc_i$ from *proc*.

      iii. Remove $proc_{i+1}$ from *proc*.

      iv. Return success.

   (b) Return failure.

This takes each consecutive tree root in *proc* and tries to MERGE them until one attempt at MERGE actually succeeds.

MOVE-SEARCH is much more complicated, as it entails searching within a tree. There are only three conditions in which a MOVE (or more than one) would be required. These condition both occur immediately after a MERGE. If α is the node that projected, and β is the other node (in other words, a copy of α became the parent of α and β), then:

1. Some of β's descendants contain features that can now be checked by moving them to α (figure 3.1).

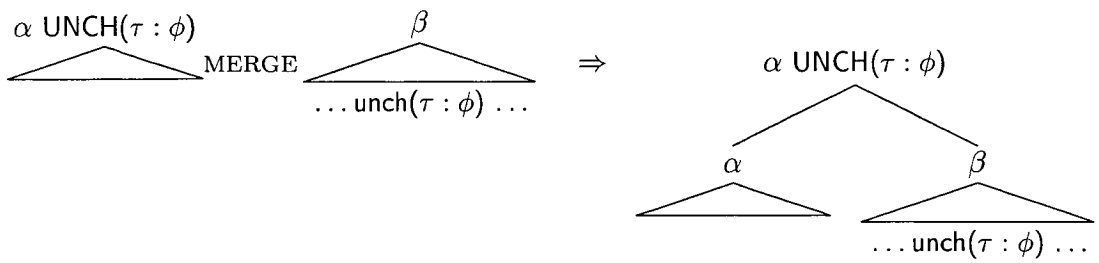

Figure 3.1: Condition 1. After the MERGE, the feature buried under β is suddenly accessible to the corresponding feature on α.

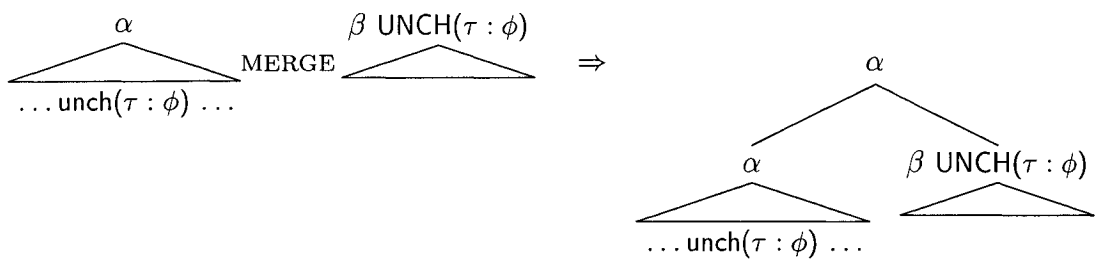

Figure 3.2: Condition 2. This time, a feature buried under α has become accessible to β.

2. Some of α's descendents, aside from β and its descendents, contain features that can be checked by moving them to β (figure 3.2).

3. If a previous MOVE caused γ to move to α or β, γ will have a similar[4] relation to α or β that α and β had with each other, possibly necessitating more MOVEs in a recursive manner (figure 3.3).

These conditions simply express what follows from the priorities given to MOVE-SEARCH and MERGE-SEARCH in OP-SEARCH. A MERGE happens in a context when there are no more subtrees to MOVE. The MERGE therefore has the potential to introduce nodes into new configurations in which movement is thereafter possible. Condition 1 states that although within β's subtree, no features may be available for MOVE, some descendents of β may now be eligible to MOVE to newly dominant α and thereafter assume a position commanding a trace left behind.

Condition 2 describes a similar situation. β now commands all of the other subtrees of α. These subtrees may now thus be eligible to move to β.

Condition 3 acknowledges that the result of a MOVE can produce more situations such as those of 1 and 2, requiring more MOVEs.

[4]It is similar in that it is governed by conditions 1 and 2.

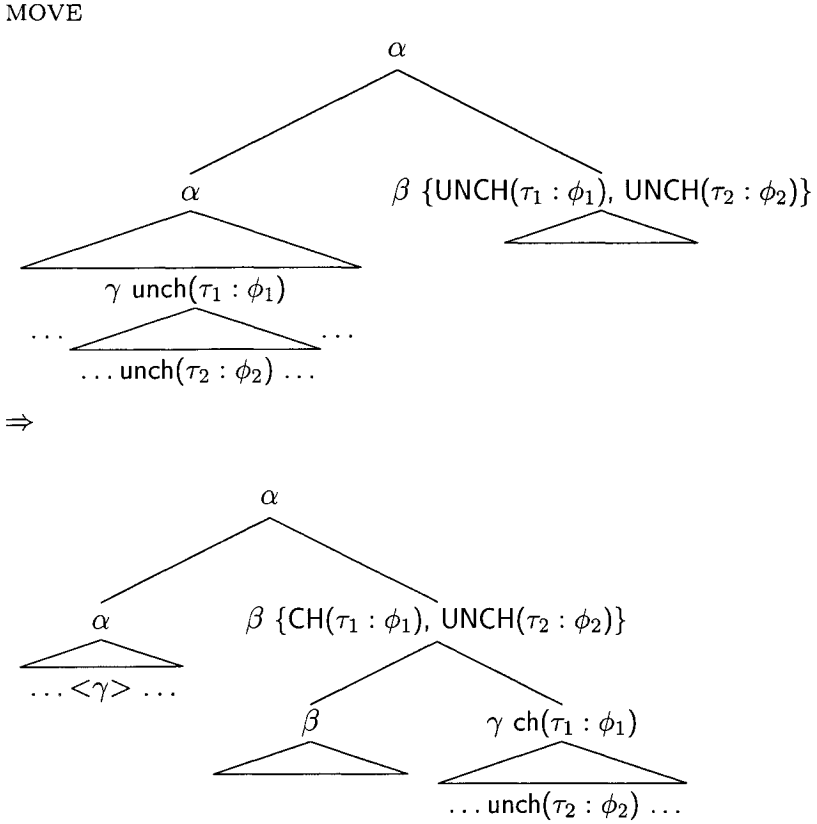

Figure 3.3: Condition 3. Moving γ created a situation where a feature under γ became accessible to a feature on β.

These conditions greatly restrict the maximal number of comparisons or attempted MOVEs that would usually need to be made between the nodes of the tree in order to find a valid MOVE. Under average conditions, only α and β need be compared each with roughly half the remaining nodes assuming that on average α and β originally dominated trees of equal size prior to MERGE. It becomes inevitably more costly in the recursive step involving γ, since if γ comes to dominate α after a MERGE with α, then the whole subtree dominated by α would need to be searched for subtrees compatible with γ.

There is the matter of the order in which the nodes of the trees are searched. Does one first attempt the left node or the right node when searching for something to MERGE with α or β? In reality, this question does not arise if we only look at maximal projections (the highest node containing the features of a given word); it makes no difference to the algorithm which child is the left child and which child is the right child. By the nature of MERGE, which also deals with maximal projections (as the root nodes of trees in the processing buffer), every pair of sister nodes contains one maximal and one intermediate projection or head. So we would thus first compare the maximal projection to α or β (depending on the conditions above), before performing the same comparison using the children of the sister node. This is rougly analogous to the Linear Correspondence Axiom of Kayne [1994] as discussed in Uriagereka [1998].

We can restrict the search to maximal projections, because maximal projections are generated by the most recent checking (by MOVE or MERGE) of the lexical item contained by the maximal projection node; consequently, the maximal projection is the node whereon the target feature for an eligible MOVE would be most likely to be exposed.

We can now begin to explicitly explain our algorithm. First, we will define a helper function, DOMINATES-MAX, which finds all the maximal projections dominated by a given node in order of precedence as defined above. Given a node n, it returns a list L containing all these nodes.

1. If n has no children, return.
2. If n's left child is a maximal projection, then let that child be known as m. Otherwise, let the right child be known as m.
3. Add m to the end of L.
4. Call DOMINATES-MAX on m.
5. Call DOMINATES-MAX on m's sister[5].

[5] m is the child of n. n could only have been created through a MERGE or MOVE, so it must have two children. So m'S sister must exist.

We can now use DOMINATES-MAX to write MOVE-SEARCH. MOVE-SEARCH takes a node and calls it α in

1. If α's left child is a maximal projection, then let that child be known as β. Otherwise, let the right child be known as β.
2. Call DOMINATES-MAX on β, the result list being L_1.
3. For each member γ_1 of L_1,
   (a) MOVE γ_1 to α.
   (b) If MOVE succeeds,
      i. Call MOVE-SEARCH on the root node of the result of MOVE.
      ii. Break loop 3.
4. Call DOMINATES-MAX on β's sister, the result being L_2.
5. For each member γ_2 of L_2,
   (a) MOVE γ_2 to β.
   (b) If MOVE succeeds
      i. Call MOVE-SEARCH on the root node of the result of MOVE.
      ii. Break loop 5.

Note that at some point, the leaf nodes will be found, in which case the for loops will have no items to loop through, and the algorithm will stop.

3.5 Extra-syntactic Considerations

3.5.1 Simple Compositional Semantics

Niyogi [2001] uses Stabler [1996] as the basis for a Minimalist parser that converts English sentences into logical representations He does this by associating a lexical entry with a λ-expression whose λs correspond to syntactic features. However, he uses Stabler's strict feature checking order. For example, if a verb needs to be checked with a subject and a complement in that order, the order of λs must be fixed in the same sequence to match the variables and fill the roles in the semantic expression.

What happens, however, when the checking order is not fixed? Then a λ-expression is not as useful. For every possible checking order for a lexical item in a free word order

language, there would have to be a separate λ-expression or some scheme to transform λ-expressions when the features corresponding to λ-bound variables are checked out of order.

We propose a scheme based on Prolog-inspired logical unification. It allows flexible processing of variable bindings that does not depend on word order. This involves expanding the lexical representation that we proposed in section 3.1.

$$\alpha : \Gamma, \Sigma$$

α and Γ are as described in section 3.1. Σ is a predicate that represents the semantic content of the word; it may contain variables.

We also revise the feature bundles:

$$\langle \beta(\tau_1 : \phi_1, \ldots, \tau_n : \phi_n) : \delta, \sigma \rangle, n \geq 1$$

β, τ, and ϕ are also as in section 3.1. σ is a variable bound to a variable in Σ. Each variable in Σ must correspond to exactly one σ in one feature bundle.

An example lexicon:

pater: ⟨unch(case:Nom, num:Sg, gnd:Masc), P⟩, patr(P)
filium: ⟨unch(case:Acc, num:Sg, gnd:Masc), F⟩, fili(F)
amat: {⟨UNCH(case:Nom, num:Sg), A⟩,
⟨UNCH(case:Acc), B⟩}, am(A, B)
laetus: ⟨unch+(case:Nom, num:Sg, gnd:Masc), X⟩, laet(X)

Niyogi [2001] uses primitives in his semantic representations, but this requires an ontology to which we are not willing to commit. It is difficult to come up with primitives for *amat* (love). We use the stems to represent whatever hypothetical primitives may apply.

We use this sentence as an example:

| laetus pater amat filium

laetus and *pater* enter and adjoin.

(pater [⟨unch(case:Nom, num:Sg, gnd:Masc), P⟩, patr(P)]
[laetus ⟨ch+(case:Nom, num:Sg, gnd:Masc), P⟩,
laet(P)]
pater)
| amat filium

The variable X in *laetus* has been unified with *pater*'s variable P. Now they refer to the same thing in their associated semantic structures. *amat* enters and MERGE is once again applied.

```
([amat {⟨CH(case:Nom, num:Sg, gnd:Masc), P⟩,
       ⟨UNCH(case:Acc), B⟩}, am(P, B)]
    ([pater ⟨ch(case:Nom, num:Sg, gnd:Masc), P⟩, patr(P)]
        [laetus ⟨ch+(case:Nom, num:Sg, gnd:Masc), P⟩,
            laet(P)]
        pater)
    amat)
| filium
```

Finally, *filium* enters and is absorbed by *amat*.

```
([amat {⟨CH(case:Nom, num:Sg, gnd:Masc), P⟩,
       ⟨CH(case:Acc, num:Sg, gnd:Masc), F⟩},
       am(P, F)]
    (amat
        ([pater ⟨ch(case:Nom, num:Sg, gnd:Masc), P⟩, patr(P)]
            [laetus ⟨ch+(case:Nom, num:Sg, gnd:Masc), P⟩,
                laet(P)]
            pater)
        amat)
    [filium ⟨unch(case:Acc, num:Sg, gnd:Masc), F⟩, fili(F)]) |
```

Now that all the features have been checked, we take all the logical expressions produced and conjoin them:

$$\text{am(P, F)} \wedge \text{laet(P)} \wedge \text{patr(P)} \wedge \text{fili(F)}$$

We do not commit to binding the variables with quantifiers because the sentence does not have any clear marking of quantification (e.g., there are no obligatory determiners in Latin).

An alternative to using logical unification is to use θ-roles. σ would instead represent roles such as AGENT and PATIENT or some alternative role-ontology. One challenge in this approach would be to develop a semantic account for adjuncts to correspond to the syntactic one. Another challenge would be to integrate these role assignments into a single representation of the meaning of the sentence. Approaches like this are more focused on a cognitively realistic account of meaning than on an account based entirely on logical formalisms.

3.5.2 Complexity

We assume that lexicon access is performed in constant time. Both forms of unification are constant-time, too[6]. So is checking a feature, and checking a bundle is bounded by the total number of feature types, thus constant again. Shifting is clearly also doable in constant time. Therefore, the dominant element is the cost of MOVE and MERGE. The problem has two parameters worth considering. The obvious one is the length of the sentence. The other size is the number of features in the processing buffer.

We look at the worst-case complexity here. Average-case complexity is a matter of future study.

The worst case for MERGE is when all of the adjacent pairs of tree roots have to be tested for compatibility until the last pair is found compatible and merged. While this is only linear in terms of words, if a pair of words has feature sets with more than one unchecked bundle, the number of tests to check for a compatible feature bundle pair is quadratic in the number of feature bundles, assuming both items have the same number of bundles, since each bundle in one item must be tested against all the bundles in the other item. Despite the pessimistic worst-case cost, it is likely that the number of features to check averages 2-3. Also, we expect that few word orders will make the parser reach the maximal number of compatibility tests, so in the end the average-case cost should be less than quadratic. Given that the number of features is related to the depth of the ultimate parse tree (in that each feature must be checked by a structure-building operation like MERGE or MOVE), this is likely to be close to $O(n \log n)$.

Determining the complexity of searching for valid MOVEs is much more difficult. The number of compatibility tests that have to be performed at each step depends on the number of nodes in all the trees in the processing buffer. It never happens that every node is tested with every other node, as a node can only be tested against its descendants or a hypothetical sister's descendants, so the running time is less than quadratically related to the number of nodes.

What is the maximum number of comparisons in the worst case? The worst case is that of the maximum number of nodes, when all the words have been MERGEd into a single tree and all remaining feature bundles checked by MOVE. Every word must be MERGEd once, and each MERGE produces one more node. If there are n words in a given sentence, there must be $n-1$ applications of MERGE to build a single tree, and thus $n-1$

[6]We assume this for the sake of argument, since we want to treat the act of checking as a fundamental operation in this system. However, given two feature bundles, if one has n features and the other has m features, then the complexity of unification is $O(nm)$ in the worst case. This is assuming that each feature in one bundle must be compared with every other feature in the other bundle.

nodes created by MERGE alone. Any remaining unchecked bundle causes one MOVE; some of these will be adjunct feature bundles that were MERGEd but not checked, so that there are at most two more nodes per feature bundle. Thus the worst-case node-size of a tree is linear in the number of features in the sentence.

The worst-case scenario for MOVE compatibility tests in that single tree happens when the tree is completely unbalanced. If the tree leans entirely to the right (all left children are leaves), then it is equivalent to all nodes being compared against all other nodes, except without duplicate pairs of nodes being compared (halving the number of comparisons) and nodes being compared with their sister (a linear reduction). This is still quadratically bounded. And since the number of nodes is linearly related to the number of feature bundles, the complexity of searches for movement is quadratically related to the number of bundles.

Given that each step consists of one search for a possible MOVE and one potential search for a possible MERGE (if no possible MOVE is found), the greater complexity of the MOVE step dominates, making each step quadratic in the total number of feature bundles. If we observe that the number of features will grow roughly linearly with the length of the input string, each step is at worst quadratic in relation to the size of the input string. Noticing that most steps involve the checking of a feature bundle, the number of steps becomes linear in the size of the input string. Assuming we overestimate each step to be the worst case, the entire parse must be better than cubically related to the size of the input. This is well within the bound provided by Harkema [2001], who determines that given n as the length of the input and k as the number of features that trigger movement, his generalized bottom-up chart recognition algorithm based on Stabler's derivational minimalism runs in time complexity $O(n^{4k+4})$.

Such a worst-case scenario may only arise during the final steps of a parse, when the number of nodes and scarcity of unchecked features reach their zenith. At that stage, it is rare that a tree is completely unbalanced. While the pessimistic cost is quadratic, we expect no worse than $O(n \log n)$ average-case complexity. Again, this is a matter of future study.

3.5.3 Lexical Effect on Complexity

In chapter 2, we discussed our focus on syntax, and in particular our assumption that the system selects correct lexical entries when required. We argued that this assumption was necessary in order to focus on the task of discovering the parsing algorithm, which must be defined prior to determining the lexical selection algorithm, as it imposes constraints

on the structure of the lexicon. However, one may wonder whether we are discarding a major factor in the total complexity of the system. We recognize at least one interesting objection in this vein: as soon as the appropriate word given the context is selected, there can be several entries for different grammatical functions of the word. Given a highly lexicalist account (such as ours), it can also be seen as a major part of the explanation for the existence of garden paths.

We propose that even if we were to consider these aspects of lexical selection in our algorithm, it would not have much more than a linear-time effect on the complexity. Consider:

- It is extremely unlikely (from a psycholinguistic standpoint) that the entire list of candidate lexical entries is traversed in order to retrieve the syntactic features. If it is not traversing the entire list, then it would be a lookup that is less than linear in time, more akin to logarithmic time.

- The worst case is that the last lexical entry for a given word is the correct one, and thus we have to backtrack on all of that word's other entries (assuming we use Prolog, but this can be quite true using equivalent mechanisms in other implementations). If there are not an absurdly large number of entries (which given the productive power of the morphological system there will not likely be), then we just get a linear-time decrease in performance.

Note, however, that this line of argumentation can be used for any parser that assumes that the human parsing system consists of multiple subsystems (once again, an assumption we mention in section 1.4). That is why it is appropriate to abstract lexical matters away from the work; for every lexical entry for a given word that we have to test, we still need to define the mechanism by which we test it! The claims regarding the parser still stand: we do not have lexical proliferation, and for each instance of an attempted parse using an incorrect lexicon entry, we still have all of the characteristics of the parser discussed in the work. Once again, an increase in complexity due to lexical ambiguity is not very relevant to the matter of defining our parsing algorithm, seeing as this problem is *always there*, and in this case, it does not even cause a very large increase in complexity.

Given the discussion above, the algorithm's complexity remains better than the worst case scenario for the most general minimalist parser—which does not really have much to say about the linguistics of discontinuous constituency or incrementalism—that so far is something like quartic [Harkema, 2001]. That is, the way we have developed our

algorithm to handle these phenomena still continues to leave room for other work to handle other phenomena, which is the most important purpose of discussing complexity at all—as opposed to the bound itself, which is less interesting.

3.6 Conclusions and Future Work

We have presented an algorithm and formal framework for the parsing of free word order languages given certain limitations: highly constrained operations with strong locality conditions (MOVE and MERGE), no attempts at simulating nondeterminism (such as lookahead), and a limitation on the availability of the words in the sentence over time (the input queue simulates a listener processing a sentence as words arrive).

The precedence of operations serves to semantically connect shifted items as soon as possible; whenever MERGE and MOVE are performed, we assume that a more complete semantic representation of the sentence is achieved. We perform the composition of meanings during the feature checking through the Prolog-style unification of variables in logical expressions associated with lexical item. The use of logical unification is to provide the flexibility that other logical variable-binding formalisms (such as the λ-calculus) do not, particularly as feature bundles in a feature set can be checked in any order. At the end of the parse, all the logical expressions are conjoined.

The algorithm favours MOVE to ensure that all features in the current set of trees in the processing buffer are exhausted. This precludes multiple pairs of compatible subtrees, since that can only happen when MERGE joins a new item to a tree without exhausting the available movement candidates.

This has the effect of creating a "shortest move condition," which Stabler [2001] enforces by only allowing one instance of a pair of compatible features in a tree. Multiple compatible pairs can only arise when a movement frees a feature set along a feature path, and the feature bundles in the feature set are each compatible with different commanded subtrees. The highest commanded subtree moves first, since it is the first found in a search from the tree root. This exception is required by the mechanisms we use to handle free word order languages, which Stabler does not consider. We conjecture that this may only appear in practice if there are actual adjunct attachment ambiguities.

The locality of MERGE (adjacent items only) precludes search for every possible pair of movement candidates on the list. MERGE's precedence over shifting and locality together have the effect of forcing MERGE of the most recently shifted items first, since they bring in new unchecked features. (This fits the intuition that semantic connections are the easiest among the most recently heard items.) Shifting last prevents occurrences of

multiple compatible candidates for merging.

We have demonstrated that the search for MERGE-eligible pairs in a particular step can be performed in linear time in relation to the number of words. The search for MOVE candidate pairs is at worst quadratic in the number of nodes, which is linear in the number of features in the sentence; cases that are close to quadratic tend to occur towards the end of a parse, and we leave it to future work to show that the average case is better than quadratic. In the worst case, the time complexity of the entire parse is less than cubic in the size of the input. Fong [2004], in his own proposal for an English minimalist parser, avoids time-consuming MOVE-candidate searches. He uses memory to store items that could move; this allows him to undo movement by reinserting the stored item at the location of a perceived trace. This technique will not work directly with free word order languages, as the structure of movement chains is more difficult to predict, but the use of memory to reduce time-complexity in our parser is an avenue for future work.

We implemented this parsing algorithm in Prolog (as we discuss in chapter 6), and we found that 90 of the 120 permutations of *pater laetus amat filium laetum* can be parsed. *filium laetus laetum amat pater* typifies the remaining 30, in that the adjective *laetum* cannot be absorbed by *amat*, since *amat* has not yet merged with *filium*; the 30 all have similar situations caused by the adjacency condition we impose on merging.

In a further experiment, we allowed implied subjects (for example, omitting *pater* is grammatical in Latin). This reduced the number of unparsable sentences to 16. *pater* was still in all the sentences, but in the 14 that became parsable, *laetus* and *pater* were originally obstructed from merging. We lifted the obstruction by allowing *laetus* to merge with *amat* first. Without implied subjects, *laetus* acted as an obstacle in the same way as *laetum*; it ceased to be an obstacle after we introduced implied subjects. We discuss some of this in the next chapter.

Before deciding how to expand the algorithm to handle the remaining 16 orders, we have embarked on a corpus study to determine whether these orders were actually plausible in classical Latin; a corpus study is necessitated by the lack of native speakers. We work with material generously provided by the Perseus Project[7]. We discuss some of this in the fifth chapter.

Work in minimalist generation and parsing has, thus far, mostly stayed within the limits of theoretical linguistics. A parser with the properties that we propose would help broaden the scope of the study of minimalist parsing to more realistic, complex linguistic phenomena. It could take this parsing philosophy toward practical applications. An

[7] `http://www.perseus.tufts.edu/`

example is speech analysis, where it would be advantageous to have a parsing algorithm that recognizes the need to make syntactic, and thus semantic, links as soon as a word enters the system.

Chapter 4

Extension: Optional Subjects

4.1 Motivation for this Chapter

In the preceding chapter, we dealt with sentences with a fully-specified subject, verb, and object. However, in some languages, particularly some highly-inflected languages such as Latin, it is possible to drop the subject of a sentence if there is a reference in the discourse. This optional nature of sentence subjects cannot be covered directly through adjunct features (unch+), since a corresponding projecting feature on the verb must be checked.

It is not only subjects that are at issue. Passive agent *by*-phrases are also optional, but the need for the agent on the verb is still there, even if the agent is unspecified in the discourse. This consideration is also true for English.

For example, *amat laetum filium* ([he] loves the happy son). The subject is missing from the sentence where it would be required in English. We can also say *amatur laetus filius* (the son is loved). In this case, we have dropped the agent phrase *a patre laeto* (by the happy father). We can also drop agent-phrases in English. But in Latin, we can drop both: *amatur* (he is loved). Nevertheless, it is a complete sentence which the algorithm described in the preceding chapter will not recognize.

4.2 Optional Subjects

We have made it a goal to avoid lexical proliferation. If we were to have a separate entry for each combination of features required to handle omitted subjects and agent phrases for a Latin passive such as *amatur*, there would be at least four entries. In order to avoid this, we create a new class of features: optional projecting features UNCH? and

CH?. During a parse, if corresponding features are found in an accessible word or tree node, UNCH? and CH? behave exactly like UNCH and CH. However, the computation does not fail if there are remaining UNCH? in the final tree; when the parser determines that no more words are available, they are automatically checked, and whatever semantic adjustments must be made to acquire a complement from the discourse are performed by those aspects of cognition that process meaning.

An extreme case in Latin would be the following: *amatur*. This single word alone can be translated to *he/she is loved (by someone)*. Here is a modified entry for *amatur* based on the original in the previous chapter:

amatur: {UNCH?(case:Nom, num:Sg, gnd:Masc), UNCH?(by:0)}

There is one optional feature for the subject and one for the agent phrase. Hence a simple parse of this one-word sentence would look like this:

| amatur

Shifting brings the word into the buffer with

[amatur {UNCH?(case:Nom, num:Sg, gnd:Masc), UNCH?(by:0)}] |

There are no more words, but two unchecked optional features remain. Thus, the system goes through the features and checks them.

[amatur {CH?(case:Nom, num:Sg, gnd:Masc), CH?(by:0)}] |

This leaves a problem: what to do when we have encountered an adjective adjacent to the verb when there is no guarantee that the corresponding noun will show up? Here is another lexicon from the previous chapter, again modified to reflect the presence of optional features:

filium: unch(case:Acc, num:Sg, gnd:Masc)
amat: {UNCH?(case:Nom, num:Sg), UNCH(case:Acc)}
laetus: unch+(case:Nom, num:Sg, gnd:Masc)
laetum: unch+(case:Acc, num:Sg, gnd:Masc)

amat has been specified as having an optional subject. We work on the sentence *filium laetus amat laetum*. So, to begin as usual:

| filium laetus amat laetum

filium and *laetus* are shifted to no immediate effect. Then so does *amat*.

```
[filium unch(case:Acc, num:Sg, gnd:Masc)]
[laetus unch+(case:Nom, num:Sg, gnd:Masc)]
[amat {UNCH?(case:Nom, num:Sg), UNCH(case:Acc)}]
| laetum
```

On the face of it, none of these can MERGE. *filium* is too far from *amat*, and due to the lack of a nominative masculine singular noun having previously merged with *amat*, *laetus* is ineligible. Then the next step would be to shift *laetum*, with the same result. The parse would thus fail.

Since the subject may not necessarily appear, we must thus specify that UNCH? features are licensed to absorb adjuncts in the way that proper CH features are. So,

```
[filium unch(case:Acc, num:Sg, gnd:Masc)]
([amat {UNCH?(case:Nom, num:Sg, gnd:Masc), UNCH(case:Acc)}]
    [laetus unch+?(case:Nom, num:Sg, gnd:Masc)]
    amat)
| laetum
```

While neither feature is checked by this MERGE, nevertheless feature unification takes place and the UNCH? feature gains a masculine gender. Why? Because once *laetus* has been absorbed into the tree, it must be guaranteed that if a noun ever comes along, it will be gender-compatible with *laetus*. Also, one should notice that unch+ has become unch+?. This also reflects the intuition that the corresponding noun may never arrive; otherwise, it behaves exactly as unch+.

The rest of the derivation proceeds as expected, leading to the final structure:

```
([amat {UNCH?(case:Nom, num:Sg, gnd:Masc), CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        <filium>
        (amat
            [laetus unch+?(case:Nom, num:Sg, gnd:Masc)]
            amat))
    ([filium ch(case:Acc, num:Sg, gnd:Masc)]
        filium
        [laetum ch+(case:Acc, num:Sg, gnd:Masc)])) |
```

Now that there are no more features to process, the parser traverses the tree, and all unchecked optional feature are checked.

```
([amat {CH?(case:Nom, num:Sg, gnd:Masc), CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        <filium>
        (amat
            [laetus ch+?(case:Nom, num:Sg, gnd:Masc)]
            amat))
    ([filium ch(case:Acc, num:Sg, gnd:Masc)]
        filium
        [laetum ch+(case:Acc, num:Sg, gnd:Masc)])) |
```

Under the scheme described in the previous chapter, there are a number of permutations that cannot be parsed. The introduction of optional subjects rescues a number of them. If we add *pater* to the end of the example sentence, the sentence remains unparsable without optional features. But with optional features, it becomes quite parsable. We will begin with the second-last tree with the extra word pater having already been shifted:

```
([amat {UNCH?(case:Nom, num:Sg, gnd:Masc), CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        <filium>
        (amat
            [laetus unch+?(case:Nom, num:Sg, gnd:Masc)]
            amat))
    ([filium ch(case:Acc, num:Sg, gnd:Masc)]
        filium
        [laetum ch+(case:Acc, num:Sg, gnd:Masc)]))
[pater unch(case:Nom, num:Sg, gnd:Masc)] |
```

Now, the MERGE:

```
([amat {CH?(case:Nom, num:Sg, gnd:Masc), CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        (amat
            <filium>
            (amat
                [laetus unch+?(case:Nom, num:Sg, gnd:Masc)]
                amat))
        ([filium ch(case:Acc, num:Sg, gnd:Masc)]
            filium
            [laetum ch+(case:Acc, num:Sg, gnd:Masc)]))
    [pater ch(case:Nom, num:Sg, gnd:Masc)]) |
```

But now there is a place for the unch+? feature to go before the whole structure is swept for optional features:

```
([amat {CH?(case:Nom, num:Sg, gnd:Masc), CH(case:Acc, num:Sg, gnd:Masc)}]
    (amat
        (amat
            <filium>
            (amat
                <laetus>
                amat))
        ([filium ch(case:Acc, num:Sg, gnd:Masc)]
            filium
            [laetum ch+(case:Acc, num:Sg, gnd:Masc)]))
    ([pater ch(case:Nom, num:Sg, gnd:Masc)]
        [laetus ch+?(case:Nom, num:Sg, gnd:Masc)]
        pater)) |
```

After this final MOVE, the parse is complete, and there are no unchecked features, optional or otherwise. Because the parser at each step only worked on the knowledge it had at the time, it absorbed *laetus* into the tree and eliminated the barrier to the parsability of a number of sentences, even though *pater* was to show up later.

4.3 Ramifications for the Semantic Formalism

The previous subsection described a scheme for allowing optional agents in Latin sentences without sacrificing the goal of preventing lexical proliferation. But what of the ramifications for a mapping between syntax and semantics? What becomes of variables attached to optional features?

Let us look at the *amatur* example, since it is simple:

```
amatur: {⟨UNCH?(case:Nom, num:Sg, gnd:Masc), F⟩, ⟨UNCH?(by:0), P⟩}, am(P, F)
```

With the single-word sentence *amatur*, both features are optional. Consequently, after the system automatically checks features at the end of the parse, we would get a semantic expression like:

```
am(P, F)
```

This is even worse than the unquantified statements in the last chapter; there is no specification of P or F at all. One could argue that this information is not required, as

additional conjuncts could be added by the semantic component in the future processing that would happen anyway. But unlike the expression resulting from a fully specified sentence with an explicit subject and object, this expression does not even carry the suggestion that more information is required.

This is a situation where higher-order logic would be applicable; where functors can be variables themselves. For every unchecked optional feature that is automatically checked, we conjoin a special unary relation to the predicate whose functor is a variable bound under some λ. We do this as part of the same process used to collect the conjuncts in the first place (see section 3.5.1). For *amatur*:

$\lambda\, X\, .\, \lambda\, Y\, .\, X(P) \wedge Y(F) \wedge am(P, F)$

This would convey to the semantic component that it is obliged to provide some referent from the discourse in order to bind all the relationships and eliminate the λs.

4.4 Conclusions and Future Work

In this section, we have actually started dealing with an issue that we have avoided mentioning explicitly through most of this thesis: the problem of empty categories. In this case, we dealt with a simple example: that which is commonly called *pro*, the invisible pronoun that is often used to represent the missing subject in optional-subject (*pro*-drop) languages [Haegeman, 1994]. We found a consistent way to represent situations such as these not by instituting complicated search and guessing mechanisms to insert *pro*, but instead to encode this situation directly upon the features that would have required *pro* using "optional" features. This solution also encodes the fact that the system would have to wait until the end of the sentence in order to be sure that the subject would not arrive.

There is still work to do. For instance, how would we handle other kinds of empty categories that are the result of more specific types of movement such as *wh*-movement whose origin, in this framework, would not be so clear as it is in many accounts of *wh*-movement in English? Since some empty categories behave as anaphora [Haegeman, 1994], how do we encode the Binding Theory into this framework, if we need to do this at all?

We have also discussed some of the ramifications of optionally-checked features for the semantic representations of sentences. Once again, as optionally checked features seem to sometimes be able to serve as substitutes for empty categories, the use of bound variable in the λ-calculus may offer a promising representation for the logical ramifications of other empty categories in this formal framework.

Chapter 5

Corpus Study

5.1 Motivation

In the previous two chapters, we discussed a grammatical formalism and an algorithm that parses most permutations of sentences exemplified by *pater laetus amat filium laetum.* However, some parses involving certain discontinuous noun phrases cannot be parsed without relaxing many of the major restrictions on the parser, such as the adjacency requirement of MERGE. In the spirit of minimalism, we consider the relaxation of the restrictions only (to allow lookahead and other such deviations from a strict sense of locality) as the last resort.

Thus we must first determine whether these sentences need to be parsed in the first place—in other words, whether these sentences are actually valid in Latin. It is usually assumed that the complex morphology and attendant agreement requirements of languages like Latin ensure that any permutation is permitted. Even native speakers of languages such as Russian may overintellectualize[1] the word order liberty they have, convincing themselves that otherwise awkward-sounding word orders that never appear may actually be valid. In addition, the lack of native speakers for a language such as Latin impedes even determining whether a word order sounds awkward.

Consequently, we decided to embark on the development of tools that aid in the analysis of Latin texts to determine what word orders we can legitimately exclude from parsability in order to minimize the changes we have to make to the parser. Our hypothesis was that the problematic word orders never occur; we designed the experiment to look for sentences that challenge this hypothesis and demonstrate that they are absent

[1] In other words, convincing themselves of facts about Russian grammar through such sources as their formal education. The possibility of this has been recognized as far back as Chomsky [1977], who makes a similar point in another context about this issue.

from the texts we examined. This section describes the data we used, the software we developed to analyze the data, and the results of our exploration. Please note that a thorough corpus examination is a much larger project than what we have undertaken so far, requiring a great deal of manual examination of texts; we are describing a pilot study that has given us preliminary results.

We would like to address one potential objection to this approach: that we are attempting to "excuse away" the orders which our algorithm cannot handle. It is valid to question whether or not our algorithm should be expected to handle certain ostensibly grammatical cases. If native speakers of certain languages do not use a grammatically correct structure, that is very likely to be a comment on its markedness. Speakers of free word order languages (such as Polish or Russian) tend to find some word orders rather highly marked. Then if something is highly marked, and a reasonable assumption of incrementality and strict assumption of locality obstructs the ability to parse it, we can reasonably say that there is some other mechanism that is going back and fixing it, which is precisely that sort of thing that makes garden paths awkward as well—having to go back and fix structure. This provides a basis for psycholinguistic experimentation, which is beyond the scope of the thesis here.

Given this, using corpus work to justify dropping certain orders is not such a bad move to make, seeing as the "excuse" is to permit incrementality and locality. Furthermore, it is quite possible that there are further mechanisms (prosodic and so on) between the compositional operators of syntax and the actual input and output; it is impossible to categorially stipulate against this possibility.

5.2 The Data

The data we used was provided by the Perseus project from compilations of speeches by M. Tullius Cicero. Cicero was chosen because he is a prose writer whose use of language was considered the most skilled among Roman writers and speakers for generations; his language is varied and complex, and he is more likely to provide the full range of plausible Latin sentences than most other writers. We list the speeches we used in appendix A.

The corpus is segmented into sentences each with its own reference code; there is no division into phrases, meaning that a sentence can have multiple complete clauses. The Perseus Project also provided a lexicon in XML containing morphological analyses of every word in this Cicero corpus. For most words, there were several analyses—massive

morphological ambiguity is the rule rather than the exception in Latin. We[2] converted this lexicon into a Prolog database for the use described in the later sections.

5.3 Methodology and Implementation

We are interested in sentences containing the following items:

1. A noun in the nominative ($\mathbf{N}_{nom}$).
2. An adjective in the nominative that agrees in gender and number with the noun in the nominative ($\mathbf{A}_{nom}$).
3. A noun in the accusative ($\mathbf{N}_{acc}$).
4. An adjective in the accusative that agrees in gender and number with the noun in the accusative ($\mathbf{A}_{acc}$).
5. A verb ($\mathbf{V}$) that agrees in person and number with $\mathbf{N}_{nom}$.

We encoded these definitions and relationships as Prolog predicates in a way that could detect the presence of these items in a list of words.

The orders that we want to exclude are:

- $\mathbf{N}_{nom}\ \mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{A}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{A}_{nom}\ \mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{N}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{V}\ \mathbf{A}_{nom}\ \mathbf{A}_{acc}\ \mathbf{N}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{N}_{acc}\ \mathbf{A}_{nom}\ \mathbf{A}_{acc}\ \mathbf{N}_{nom}\ \mathbf{V}$
- $\mathbf{N}_{acc}\ \mathbf{A}_{nom}\ \mathbf{N}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}$
- $\mathbf{N}_{acc}\ \mathbf{N}_{nom}\ \mathbf{A}_{acc}\ \mathbf{A}_{nom}\ \mathbf{V}$
- $\mathbf{N}_{nom}\ \mathbf{N}_{acc}\ \mathbf{A}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}$
- $\mathbf{N}_{acc}\ \mathbf{N}_{nom}\ \mathbf{A}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}$
- $\mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{N}_{nom}\ \mathbf{N}_{acc}\ \mathbf{A}_{nom}$
- $\mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{A}_{nom}\ \mathbf{N}_{acc}\ \mathbf{N}_{nom}$

[2]Much of the programming work was done by Michael Duda under Asad Sayeed's immediate direction.

- $\mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{A}_{nom}\ \mathbf{N}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{A}_{nom}\ \mathbf{N}_{acc}\ \mathbf{N}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}$
- $\mathbf{V}\ \mathbf{A}_{acc}\ \mathbf{N}_{nom}\ \mathbf{A}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{N}_{acc}\ \mathbf{N}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}\ \mathbf{A}_{nom}$
- $\mathbf{V}\ \mathbf{N}_{nom}\ \mathbf{A}_{acc}\ \mathbf{A}_{nom}\ \mathbf{N}_{acc}$
- $\mathbf{N}_{acc}\ \mathbf{A}_{nom}\ \mathbf{A}_{acc}\ \mathbf{V}\ \mathbf{N}_{nom}$

We call permutations of the five types "sentence classes." We usually call the above "problematic sentence classes" in order to emphasize their undesirability. What do these sixteen have in common? In all of them, there is an $\mathbf{A}_{acc}$ somewhere between a nominative form and a verb without also being next to a $\mathbf{N}_{acc}$. $\mathbf{A}_{acc}$ forms are clearly not able to MERGE with nominative forms, and they are also not able to MERGE with $\mathbf{V}$ forms, unlike $\mathbf{A}_{nom}$ forms for reasons described in chapter 4—we have specified that verbs take optional subjects, not optional objects. This $\mathbf{A}_{acc}$ form obstructs the nominative form from becoming adjacent to the verb in order to permit MERGE; however, were it next to its corresponding $\mathbf{N}_{acc}$, it would MERGE with the $\mathbf{N}_{acc}$, which can in turn MERGE with $\mathbf{V}$, no longer obstructing the nominative forms from also merging with the verb.

Our overall process was the following. For each sentence in the corpus:

1. Strip the punctuation.
2. Break the string into words.
3. Find the first five words that satisfy the five types above.
4. Find all the type-assignments for these words. (There may be many due to morphological ambiguity.)
5. Determine to which sentence class each type-assignment belongs. For each type-assignment:
   (a) Determine if it belongs to a problematic sentence class.
   (b) If so, add the type-assignment (consisting of the words used associated with one of the above types) and the reference code of the sentence to which the type-assignment belongs to the list of sentences in that class (which is written out to a file).

We could have chosen every set of five consecutive relevant words in the sentence. But this would have vastly increased the number of situations in which a five-word set would cross a clause boundary. Already, with only the first five relevant words being considered, there are many instances where the words cross the clause boundary. Having effectively a five-word "window" traversing the sentences would increase this many times, drowning out any potentially-existing valid examples of these undesirable sentence classes. The only way to go meaningfully beyond the restriction is to develop robust Latin clause chunking. The restriction to the first five relevant words at least provides an increased likelihood that all five will belong to the same clause.

Since many of these five-word sets cross clause boundaries, there may be many spurious identifications of sentences in problematic classes; likewise, morphological ambiguity contributes to this. We are, in fact, hypothesizing that *all* of them are spurious. Consequently, we needed to examine any that appear after the algorithm is run on the texts and identify them as incorrectly classifed; or we needed to do this to enough of them that we would be relatively confident that we would not encounter a genuine example of a sentence in a problematic class.

Given that the Cicero corpus is quite large, how would we compute the number of type-assignments we need to check to get an acceptable confidence interval and level? To do this, we make use of the formula for sample size determination:

$$n = \left(\frac{z}{\delta}\right)^2 \hat{\Pi}(1 - \hat{\Pi}) \tag{5.1}$$

where n is the sample size, z a value related to the confidence level required (usually 95%, giving a value of 1.96), δ is the confidence interval, and $\hat{\Pi}$ is the proportion of the sample expected to give a certain value [Mansfield, 1991] (in our case, whether or not the type-assignment was spurious). We can also turn it around and solve for the confidence interval:

$$\delta = z\sqrt{\frac{\hat{\Pi}(1 - \hat{\Pi})}{n}} \tag{5.2}$$

We then randomly select n type-assignments from the population and examine whether they are valid assignments given the structure and context of the sentences to which they are assigned. This step is done manually and is very time-consuming; it requires a careful examination not only of the sentence in question, but of potentially many sentences around it, in order to determine whether a word in a sentence has been provided the correct type given the massive type-ambiguity that can exist in a Latin sentence.

If we ever encounter a single clear example of an undesirable sentence, then this investigation can, in theory, stop; and we can say that sentences in the undesirable classes exist in Latin, and in future work, we would have to find a way to include them. But is this really so? Perhaps, if it turns out that there are an extremely small number of sentences in undesirable classes compared to other classes, we could still attribute this to other factors, such as stylistic factors or even a simple mistake.

If we do not encounter any examples of an undesirable sentence in our sample, then obviously we cannot *directly* assume that there aren't any in the corpus, or that it is completely ungrammatical. However, using the formulae above, we would be able to compute the maximum likelihood (δ) that we might encounter one in the future, given that all of the previous did not belong to the undesirable classes. If that likelihood is low, we would be able to build a case that a parser reflective of human linguistic competence need not be able to parse these sentences.

5.4 Results and Analysis

Though the process of collecting the results was fairly complicated, the results themselves are quite simple:

- In the corpus, there were 20,082 sentences in total, sentences often composed of many clauses.
- Of these, 18,414 had more than five words.
- Of these, 16,719 did not contain the requisite five types. This means that 1,695 sentences contained the five types.
- These 1,695 sentences produced 5190 possible type assignments in total.
- Of these, only 356 were *potentially* of the undesirable sentence classes.

In other words, only a small percentage (9.2%) of the sentences longer than five words were actually relevant to this study in containing the five word types. Of those, only 21% of the sentences containing those word types were potential members of undesirable sentences classes (compared with the undesirable sentence classes that make up 16/120 = 13.3% of the total number of sentence classes).

We examined 176 of the analyses. All of them were spurious. Here is an example of a sentence for which a spurious analysis was found:

> *ac si* **quis est talis qualis** *esse* **omnis** *oportebat, qui in hoc ipso in quo exsultat et triumphat oratio mea me vehementer accuset, quod ta m capitalem hostem non comprehenderim potius quam emiserim, non est ista mea cul pa, Quirites, sed temporum.* (Cic. Catil. 2.3)

The words in bold are those that were identified as being the first words in the sentence that could potentially fit an undesirable analysis.

- *quis* is $\mathbf{N}_{nom}$.
- *est* is **V**.
- *talis* is $\mathbf{A}_{acc}$.
- *qualis* is $\mathbf{A}_{nom}$.
- *omnis* is $\mathbf{N}_{acc}$.

Given the parsing algorithm described in this work, we can see that neither adjective can reach its corresponding noun; they entirely block each other. But this is only true assuming that this type-assignment is correct. In reality, *talis* and *qualis* (together, "of such a nature") are intended to be of the same case, not different cases; it so happens that for both of them their nominative and accusative cases are identical. So the example is spurious and can be crossed off the list.

Since we have examined a significant number of such examples selected at random, we can now make use of the formula for δ to find the confidence interval given a confidence level of 95%. Since all of them were spurious, we can make $\hat{\Pi} = 1$ (100%). This makes $1 - \hat{\Pi} = 0$. So $\delta = 0$; this result, though extremely desirable, is however probably not a useful representation of the situation. It predicts a perfect relationship between the sample data and the actual population partly by assuming that the lone human sentence-classifier made no mistakes in the rejection of all the undesirable analyses. To resolve this, we must assume a small amount of error on the part of the human doing the classification; in which case, if we assume that 1% of the spurious analyses might actually be real (even though the human classifier did not report this), then we actually get a confidence interval of 1.47%.

The actual size of the population is 356, so we sampled and classified about half of them. In that case, the actual interval given a 95% confidence level and assuming that (although the human classifier found 100% of the sample to be spurious) we will use 99% as the number that we are certain must be spurious would probably be less than 1.47%.

5.5 Conclusions and Future Work

We have demonstrated the outline of a method to substitute partially automated studies of corpora for grammaticality judgements for use when grammaticality judgements are impossible to obtain, such as for ancient language like Latin. We applied it to some of the works of Cicero to determine whether or not limitations on the kinds of noun phrase discontinuities accepted by the parser described in the third and fourth chapters would be detrimental to the parser's coverage of Latin sentences. We found that at least in the Cicerine corpus that we used, it was unlikely that the sentences we were excluding from the parser would actually appear.

This method substitutes the judgement of a non-native speaker as to whether a number of sentences fit into a sentence class for the judgement of a native speaker who can give, in theory, a clear and direct response as to the grammaticality of a sentence. However, one vital element is missing: an assessment of the accuracy of the non-native-speaker evaluator. Due to resource constraints, we were only able to employ one evaluator (it is difficult to find people proficient in Latin willing to perform the time-consuming evaluation tasks); otherwise, we could estimate the human error in this process by comparing the results of multiple independent evaluators and mutual agreement (κ).

Specific to this experiment, we used only some of the material from a single Latin author. Selecting a single orator such as Cicero with interestingly varied prose allows us to use this technique to simulate the competence of a Latin speaker without having to worry about language variation over dialect and time, which may have been the result of using multiple authors. Nevertheless, it is necessary to expand the corpus so that we can find more than 356 eligible sentences to sample and test for membership in the problematic classes. This we may be able to do by including some of Cicero's contemporaries, such as Julius Caesar. To ensure maximum correspondence in their internal knowledge of Latin, we would have to examine the biographies of such authors so as to ensure that they would have absorbed Latin in roughly the same linguistic environments.

It remains a somewhat philosophical problem as to whether the results obtained from such investigations actually reflect linguistic competence in such a way that a parser based on them can be held to be also a reflection of a native human parser. After all, it could be that some of the sentences that do not appear in corpora are simply unrealized potentials of the linguistic competence of the writer. Nevertheless, for a "dead" language, it is necessary to accept that we may never know the answer to this; we are, however, confident that in the specific parts of Latin syntax on which we have decided to focus, our corpus investigation has brought us to the closest approximation of that knowledge

that we are likely ever to get.

One could raise another objection at this point: the use of a data-driven investigation such as this corpus study might be undermining the competence/performance distinction that is central to this and other work in generative grammar. This could be used as a general complaint against any attempt to align corpus statistics with the predictions of a Chomskyan parsing algorithm. But the competence/performance dichotomy is not in itself damaged by this kind of work as long as one is willing to admit that competence has an effect on performance and thus influences the distribution observed sentences. Abney [1996] has a discussion that defends quite eloquently the notion that competence is a system that biases the distribution of highly variable events. In which case, this particular objection does not hold: it is not a contradiction to make use of corpus-based investigations when exploring matters of linguistic competence.

A simpler way to put it is that even in minimalist-inclined circles, the parsing/generation system is often now treated as a potential generator of a distribution, and it does not delegitimize the competence/performance distinction in itself to compare to an approximation of that distribution as represented by actual texts. We are not saying, after all, that learning can be represented under some naive Markov model variant, which is the main objection to incorporating various stochastic techniques into approaches to syntax based on generative grammar.

Chapter 6

Implementation

6.1 Introduction

In order to demonstrate the feasibility of the parses we presented in the third and fourth chapters, we implemented two versions of the algorithm in SWI Prolog 5.4.4[1]. We provide the code for the version based on chapter 3 in appendix B and the chapter 4 extension for optional subjects in appendix C. In addition, in appendix B we also provide output from the program given the four sentences in chapter 3; these demonstrate that our implementation of the parser provides the predicted derivations as output. Likewise, in appendix C, we provide the output of the chapter 4 version of the parser given the two sentences we examined in chapter 4. In both appendices, we also include the derivations of additional sentences.

The parser that we implemented does not include the compositional semantics that we briefly discussed in chapters 3 and 4. The discussion of semantics was largely intended to demonstrate the potential of this parser to be integrated into a larger view of linguistics beyond syntax. Few ramifications of the semantics presented for the operation of the parser were discussed in detail, and consequently we did not implement it. Similarly, while we did implement optional subjects as in the fourth chapter, there are also ramifications for optional feature checking in general that we did not explore, and hence we did not explore more types of optional structures than the optional subjects we discussed.

We implemented the parser in Prolog in order to take advantage of a few of the benefits of the language. Prolog's reliance on recursion, its use of unification, and the simplicity in its manner of representing and processing lists make the searching of trees and the

[1]SWI Prolog is available at `http://www.swi-prolog.org/`. In addition, the code was tested on a slightly earlier version of SWI Prolog (5.2.13), and it should function in a similar manner on versions at least as early as 5.2.x, if not earlier.

unification of feature structures very easy to implement. Its declarative nature is useful in that it ideally suits the manner in which the algorithm of chapter 3 is described, and as new conditions are added to the system, it is easy to specify them without making significant alterations to the code. Finally, the process of backtracking enables future research in this area in that it can be used to produce a series of ranked alternative parses, thereby enabling potential work in phrase attachment ambiguities.

6.2 Source Code Structure

We divided the source code into several files as listed in appendices B and C. We describe them briefly below:

parser.pl This contains the main loop of the program. It requests the next step in the derivation and prints it out until there are no more possible steps. In the version with optional subjects in appendix C, it also performs the final step of checking unchecked optional features.

deriv.pl This file determines the next step in the parse by attempting each of the operations MOVE and MERGE and returning the result of the one that succeeds. Otherwise, it shifts the next word into the processing buffer of the derivation.

ops.pl This defines the operations MOVE and MERGE in terms of searches for tree nodes that have feature bundles that are compatible with one another for checking.

check.pl This file is central to the implementation. It defines the process of checking feature paths in terms of unification, and it deals with all of the possible cases under which this might occur. It also determines which nodes project. It has a separate version for optional subjects in appendix C.

unify.pl This file contains the implementation of feature set unification that forms the basis for feature checking.

portrayal.pl This updates the display predicates of SWI Prolog with the information required to print out the derivations in a form similar to that of the third and fourth chapters. It has a slightly modified version in appendix C that prints out optionally checked features correctly.

latin.lex.pl This contains a small Latin lexicon, sufficient for demonstrating the parses provided in chapter 3. It is modified to allow the verbs to take optional subjects in appendix C.

english.lex.pl This file contains a small English lexicon, also for demonstrating the parses provided in chapter 3.

6.3 Data Structures

We implement the tree structure that we use in our algorithm with very simple Prolog terms representing nodes, as follows:

```
maxnode(Label, Features, LeftChild, RightChild)
node(Label, Features, LeftChild, RightChild)
trace(Label)
none
```

A maxnode is a maximal projection as described in the third chapter. It contains a Label, which is simply the word in the sentence as it appears in the lexicon. It contains a list of Features which are represented as Prolog terms that are very similar to the feature bundle representation in chapter 3. And it has a LeftChild and a RightChild that are also nodes.

A node is exactly the same as a maxnode except that it is not a maximal projection. A trace keeps only the Label of the node that was moved from its position. A leaf maxnode or node has children that are set to none.

6.4 A Note on Performance

We observed the performance of the chapter 3 implementation of the parser using SWI Prolog's time/1 predicate in order to determine the number of inferences and the actual running time of the program given the example sentences in chapter 3. The tests were performed on an Intel Pentium M 1.5Ghz running Mandrake Linux 10.0.

Sentence	**Inferences**	**Time (sec.)**
pater laetus filium amat laetum	50,272	0.05
pater laetus a filio laeto amatur	17,553	0.02
the happy father loves the happy son	29,403	0.02
the son is loved by the father	25,401	0.02
the happy son is loved by the happy father	53,787	0.08

Table 6.1: The performance of example parses from chapter 3.

The results are in table 6.1. We included an extra sentence, *the happy son is loved by the happy father*, in order to make sure that our comparison of the English sentences was fair: we needed to add the adjectives.

One striking observation is how much faster the parsing of the Latin passive example is than the active. However, this is easily explained: the Latin active example includes a discontinuous constituency. *pater laetus filium amat* form a tree before *laetum* is shifted. Our implementation searches the tree for the appropriate noun to MOVE and MERGE with *laetum*—this creates many more attempts at checking in the Prolog implementation.

This highlights a drawback of making use of Prolog's natural recursion. While the appriopriate candidate for movement is always the highest one in our algorithm, because the recursion passes down two separate branches of computation, the program searches the whole tree rather than interrupting when the first candidate is found.

This does not explain why the introduction of *happy* in the English examples greatly increases the number of inferences performed. A good explanation for this lies perhaps in the simplicity of the features involved in the English example: every adjective and article provokes a search of the tree for type:N features. These searches prove to be ultimately fruitless, but they produce a large number of inferences before failing. We leave further exploration of the efficiency of the many possible combinations of words and features given the algorithm and various extensions to it for future work.

One major area in which performance can be improved is in the use of external memory for coordinating branches of recursion as discussed above and for memoizing the results of unifications, as the code often contains repeated unification checks for different stages of MERGE. Saving unification results for a limited amount of time will reduce the number of inferences while perhaps preserving some form of psychological realism.

Lastly, it should be noted much of the time in seconds taken by the parses is actually the cost of printing the steps.

Chapter 7

Final Concluding Remarks

In the first chapter, we discussed a number of philosophical and methodological issues underlying this project, such as the factors to take into consideration when parsing free word order languages, and the difference between parsing and simply reversing generation. In the second chapter, we discussed the details of our specific focus on Latin and minimalism, describing a prior and well-known system of formal minimalist syntactic representation. The third chapter presented a new parsing algorithm that takes into account most of the factors discussed in the first and second chapters. The fourth chapter demonstrated a way to handle optional subjects in Latin by slightly modifying the system described in the third chapter; allowing optional subjects also provided an elegant way to eliminate barriers to the parsability of other sentences not handled by the third chapter's algorithm. The fifth chapter described a corpus experiment that suggested a way to justify some of the limitations on the parser. Finally, the sixth chapter discussed the work we did in implementing our parsing algorithm.

Each chapter came to its own conclusions and described some future directions for the branches of research into this topic that it represented. Therefore, we will not unnecessarily reiterate them here. Nevertheless, we believe that we have provided the basis for a program of research into questions that were hitherto quite difficult in the study of free word order languages, particularly those that are ancient languages.

Bibliography

Steven Abney. Statistical methods and linguistics. In Judith Klavans and Philip Resnik, editors, *The Balancing Act: Combining Symbolic and Statistical Approaches to Language*. The MIT Press, Cambridge, Massachusetts, 1996.

Noam Chomsky. *Aspects of the Theory of Syntax*. MIT Press, Cambridge, MA, 1965.

Noam Chomsky. On wh-movement. In Peter Culicover, Thomas Wasow, and Adrian Akmajian, editors, *Formal Syntax*, pages 71–132. Academic Press, New York, 1977.

Noam Chomsky. *The Mimimalist Program*. MIT Press, Cambridge, MA, 1995.

Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, 1991.

Sandiway Fong. Computation with probes and goals: A parsing perspective. In A. M. Di Sciullo and R. Delmonte, editors, *UG and External Systems*. John Benjamins, Amsterdam, 2004.

Werner Frey and Hans-Martin Gärtner. On the treatment of scrambling and adjunction in minimalist grammars. In G. Jäger, P. Monachesi, G. Penn, and S. Wintner, editors, *Proceedings of Formal Grammar*, 2002.

Liliane Haegeman. *Introduction to Government and Binding Theory*. Blackwell, 2 edition, 1994.

Hendrik Harkema. *Parsing Minimalist Languages*. PhD thesis, University of California at Los Angeles, 2001.

Jarosław Jakielaszek. *Raising under control: the case of Latin*. University of Warsaw, Warsaw, 2003.

Aravind K. Joshi. Tree adjoining grammars: How much context-sensitivity is required to provide reasonable structural descriptions. In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing: Psychological, computational, and theoretical perspectives*, Cambridge, MA, 1985. Cambridge University Press.

Aravind K. Joshi, K. Vijay-Shanker, and David Weir. The convergence of mildly context-sensitive grammar formalisms. In Peter Sells, Stuart M. Shieber, and Thomas Wasow, editors, *Foundational Issues in Natural Language Processing*, Cambridge, MA, 1991. MIT Press.

Michael B. Kashket. Parsing Warlpiri—a free word order challenge. In Robert C. Berwick, Steven P. Abney, and Carol Tenny, editors, *Principle-Based Parsing: Computation and Psycholinguistics*. Kluwer, Dordrecht, 1991.

Richard Kayne. *The Antisymmetry of Syntax*. MIT Press, Cambridge, MA, 1994.

Matthias Trautner Kromann. Toward discontinuous grammar formalisms with lexical notions of valency. Master's thesis, Copenhagen Business School, 1999.

Edwin Mansfield. *Statistics for Business and Economics*. W. W. Norton, New York, 4 edition, 1991.

Jens Michaelis. Derivational minimalism is mildly context-sensitive. *Lecture Notes in Computer Science*, 2014, 2001.

Shigeru Miyagawa. Against optional scrambling. *Linguistic Inquiry*, 28(1):1–25, 1997.

Sourabh Niyogi. A minimalist interpetation of verb subcategorization. International Workshop on Parsing Technologies (IWPT-2001), 2001.

Gerald Penn and Mohammad Haji-Abdolhosseini. Topological parsing. Budapest, 2003. EACL-03.

Harm Pinkster. *Latin Syntax and Semantics*. Routeledge, New York, 1990. Translated by Hotze Mulder.

O. Rambow and Y.-S. Lee. Word order variation and tree adjoining grammars. *Computational Intelligence*, 10:386–400, 1994.

Asad Sayeed and Stan Szpakowicz. Developing a minimalist parser for free word order languages with discontinuous constituency. In José Luis Vicedo, Patricio Martínez-Barco, Rafael Muñoz, and Maximiliano Saiz, editors, *EsTAL—España for Natural Language Processing*. Springer-Verlag, 2004.

Irina Sekerina. Scrambling and processing: Dependencies, complexity, and constraints. In S. Karimi, editor, *Scrambling and Word Order*, Malden, MA, 2002.

Edward P. Stabler. Derivational minimalism. Logical Aspects of Computational Linguistics (LACL 96), 1996.

Edward P. Stabler. Minimalist grammars and recognition. In Christian Rohrer, Antje Roßdeutscher, and Hans Kamp, editors, *Linguistic Form and its Computation*. CSLI Publications, Stanford, 2001.

Juan Uriagereka. *Rhyme and Reason: An Introduction to Minimalist Syntax*. MIT Press, Cambridge, Mass., 1998.

Appendix A

The Corpus

In chapter 5, we mentioned that we used compilations of some of Cicero's speeches for our corpus study. We list them in this appendix.

The speeches were all compiled and edited by Albert Clark:

- *Orationes: Cum Senatui gratias egit, Cum populo gratias egit, De domo sua, De haruspicum responso, Pro Sestio, In Vatinium, De provinciis consularibus, Pro Balbo*
- *Orationes: Divinatio in Q. Caecilium, In C. Verrem*
- *Orationes: Pro Milone, Pro Marcello, Pro Ligario, Pro rege Deiotaro, Philippicae I-XIV*
- *Orationes: Pro P. Quinctio, Pro Q. Roscio comoedo, Pro A. Caecina, De lege agraria contra Rullum, Pro C. Rabiro perduellionis reo, Pro L. Flacco, In L. Pisonem, Pro C. Rabiro Postumo*
- *Orationes: Pro Sex. Roscio, De imperio Cn. Pompei, Pro Cluentio, In Catilinam, Pro Murena, Pro Caelio*
- *Orationes: Pro Tullio, Pro Fonteio, Pro Sulla, Pro Archia, Pro Plancio, Pro Scauro.*

Appendix B

The Program—Chapter 3 version

B.1 Source Code

B.1.1 parser.pl

```
% This file is the main loop of the program.  It goes through
% each step and prints it out, numbered.
% You have to load a lexicon file before you use this.

:- use_module(deriv).
:- [portrayal].

parse(A) :-
        do_parse(deriv([], A), 0).

% Every call to step is cut so that only the first solution is taken.
do_parse(Deriv, 0) :-
        nl, write('Initial state:'),
        print(Deriv),
        step(Deriv, NextDeriv), !,
        do_parse(NextDeriv, 1).
do_parse(Deriv, N) :-
        N > 0,
        nl, nl, write('Step '), write(N), write(':'),
        print(Deriv),
        step(Deriv, NextDeriv), !,
        M is N + 1,
        do_parse(NextDeriv, M).
do_parse(_, _) :-
        nl, nl, write('That was the last possible step.').
```

B.1.2 deriv.pl

```
:- module(deriv, [step/2]).
:- use_module(ops).
```

```
% step performs the next step in the derivation.
step(deriv(Processing, Input), deriv(NewProcessing, Input)) :-
        next_move(Processing, NewProcessing).
step(deriv(Processing, Input), deriv(NewProcessing, Input)) :-
        next_merge(Processing, NewProcessing).
step(deriv(Processing, Input), deriv(NewProcessing, NewInput)) :-
        shift(Processing, Input, NewProcessing, NewInput).

% Now we find the next move.
next_move([Tree | Rest], [NewTree | Rest]) :-
        move(Tree, NewTree).
next_move([Tree | Rest], [Tree | NewRest]) :-
        next_move(Rest, NewRest).

% We find the next merge.
next_merge([Tree1, Tree2 | Rest], [NewTree | Rest]) :-
        mmerge(Tree1, Tree2, NewTree).
next_merge([Tree | Rest], [Tree | NewRest]) :-
        next_merge(Rest, NewRest).

% Shift something from the input queue.
shift(Processing, [Next | InputRest], NewProcessing, InputRest) :-
        lex(Next, Entry),
        append(Processing, [maxnode(Next, Entry, none, none)],
                NewProcessing).
```

B.1.3 ops.pl

```
:- module(ops, [mmerge/3, move/2]).
:- use_module(check).

% mmerge can't be called merge because that conflicts with a SWIPL builtin.
% So, mmerge for "minimalist merge".  check is deliberately asymmetric to
% allow us to distinguish between the projecting and non-projecting nodes.
% So we need two clauses for merge to deal with the asymmetry (cuts down
% vastly on code to do it this way.
%
% We indicate maximal projections with "maxnode" and intermediate projections
% and heads with "node."  In reality we only work with maximal projections,
% so we need not store the bundles in nonmaximal projections, but we do
% for debugging purposes.  They won't be printed out.
mmerge(maxnode(A, ABundles, ALeftChild, ARightChild),
       maxnode(B, BBundles, BLeftChild, BRightChild),
       maxnode(A, AFixedBundles,
               node(A, AFixedBundles, ALeftChild, ARightChild),
               maxnode(B, BFixedBundles, BLeftChild, BRightChild))) :-
        member(AUnchecked, ABundles),
        member(BUnchecked, BBundles),
        check(AUnchecked, BUnchecked, AChecked, BChecked, ADirection,
```

```
            BDirection),
        valid_direction(ADirection, BDirection),
        delete(ABundles, AUnchecked, InterABundles),
        AFixedBundles = [AChecked | InterABundles],
        delete(BBundles, BUnchecked, InterBBundles),
        BFixedBundles = [BChecked | InterBBundles].
mmerge(maxnode(A, ABundles, ALeftChild, ARightChild),
       maxnode(B, BBundles, BLeftChild, BRightChild),
       maxnode(B, BFixedBundles,
               maxnode(A, AFixedBundles, ALeftChild, ARightChild),
               node(B, BFixedBundles, BLeftChild, BRightChild))) :-
        member(AUnchecked, ABundles),
        member(BUnchecked, BBundles),
        check(BUnchecked, AUnchecked, BChecked, AChecked, BDirection,
            ADirection),
        valid_direction(ADirection, BDirection),
        delete(ABundles, AUnchecked, InterABundles),
        AFixedBundles = [AChecked | InterABundles],
        delete(BBundles, BUnchecked, InterBBundles),
        BFixedBundles = [BChecked | InterBBundles].

% We need to enforce proper directionality here.
valid_direction(both, both).
valid_direction(right, left).
valid_direction(right, both).
valid_direction(both, left).

% time for move.  There's a first nonrecursive layer to pick candidates.
% Cases where the move target is the root.
move(maxnode(A, B, LeftChild, RightChild),
     NewMax) :-
        LeftChild =.. [maxnode | _],
        find_candidate(maxnode(A, B, _, _), LeftChild, NewLeftChild,
                       Candidate, 0, _),
        mmerge(Candidate, maxnode(A, B, NewLeftChild, RightChild), InterMax),
        (   move(InterMax, NewMax)
        ;   NewMax = InterMax).
move(maxnode(A, B, LeftChild, RightChild),
     NewMax) :-
        RightChild =.. [maxnode | _],
        find_candidate(maxnode(A, B, _, _), RightChild, NewRightChild,
                       Candidate, 0, _),
        mmerge(maxnode(A, B, LeftChild, NewRightChild), Candidate, InterMax),
        (   move(InterMax, NewMax)
        ;   NewMax = InterMax).
% Cases where the move target is the maxnode child.
move(maxnode(A, B, LeftChild, RightChild),
     maxnode(A, B, NewLeftChild, NewRightChild)) :-
        LeftChild =.. [maxnode | _],
```

```
        % the other MUST be a non-maximal.
        find_candidate(LeftChild, RightChild, NewRightChild, Candidate, 0, _),
        mmerge(LeftChild, Candidate, InterNewLeftChild),
        (   move(InterNewLeftChild, NewLeftChild)
        ;   NewLeftChild = InterNewLeftChild).
move(maxnode(A, B, LeftChild, RightChild),
     maxnode(A, B, NewLeftChild, NewRightChild)) :-
        RightChild =.. [maxnode | _],
        % the other MUST be a non-maximal.
        find_candidate(RightChild, LeftChild, NewLeftChild, Candidate, 0, _),
        mmerge(Candidate, RightChild, InterNewRightChild),
        (   move(InterNewRightChild, NewRightChild)
        ;   NewRightChild = InterNewRightChild).

% find_candidate searches the tree for a candidate for movement given
% a location to move to.  It provides the candidate and the tree without
% the candidate (replaced by a trace).
% What we really want is a breadth-first search but that is awkward, so
% we use a depth-first search and compare depths.
% We first deal with the base cases: we've found something.
find_candidate(Higher, Tree, NewTree, LeftChild, Level, Level) :-
        functor(Tree, Type, _),
        member(Type, [node, maxnode]),
        Tree =.. [Type, A, B, LeftChild, RightChild],
        mmerge(Higher, LeftChild, _),
        LeftChild =.. [_, Name, _, _, _],
        NewTree =.. [Type, A, B, trace(Name), RightChild].
find_candidate(Higher, Tree, NewTree, RightChild, Level, Level) :-
        functor(Tree, Type, _),
        member(Type, [node, maxnode]),
        Tree =.. [Type, A, B, LeftChild, RightChild],
        mmerge(Higher, RightChild, _),
        RightChild =.. [_, Name, _, _, _],
        NewTree =.. [Type, A, B, LeftChild, trace(Name)].
% Now we need to try the recursive cases. First, what if only one
% branch were eligible for recursion (the other being a trace).
find_candidate(Higher, Tree, NewTree, Candidate, Level, NewLevel) :-
        Tree =.. [Type, A, B, LeftChild, trace(Name)],
        InterLevel is Level + 1,
        find_candidate(Higher, LeftChild, NewLeftChild, Candidate,
                       InterLevel, NewLevel),
        NewTree =.. [Type, A, B, NewLeftChild, trace(Name)].
find_candidate(Higher, Tree, NewTree, Candidate, Level, NewLevel) :-
        Tree =.. [Type, A, B, trace(Name), RightChild],
        InterLevel is Level + 1,
        find_candidate(Higher, RightChild, NewRightChild, Candidate,
                       InterLevel, NewLevel),
        NewTree =.. [Type, A, B, trace(Name), NewRightChild].
% The last case occurs when both the left and the right sides are valid
```

```
% subtrees, and we have to use the level numbers to break the tie.
% This involves some subcases, but we'll see.
find_candidate(Higher, Tree, NewTree, Candidate, Level, NewLevel) :-
        Tree =.. [Type, A, B, RightChild, LeftChild],
        InterLevel is Level + 1,
        find_candidate(Higher, LeftChild, NewLeftChild, LeftCandidate,
                       InterLevel, LeftLevel),
        find_candidate(Higher, RightChild, NewRightChild, RightCandidate,
                       InterLevel, RightLevel),
        % left is higher given lower level
        (   LeftLevel =< RightLevel
        ->  NewTree =.. [Type, A, B, NewLeftChild, RightChild],
            NewLevel = LeftLevel,
            Candidate = LeftCandidate
        ;   NewTree =.. [Type, A, B, LeftChild, NewRightChild],
            NewLevel = RightLevel,
            Candidate = RightCandidate).
find_candidate(Higher, Tree, NewTree, Candidate, Level, NewLevel) :-
        Tree =.. [Type, A, B, LeftChild, RightChild],
        InterLevel is Level + 1,
        find_candidate(Higher, LeftChild, NewLeftChild, Candidate,
                       InterLevel, NewLevel),
        NewTree =.. [Type, A, B, NewLeftChild, RightChild].
find_candidate(Higher, Tree, NewTree, Candidate, Level, NewLevel) :-
        Tree =.. [Type, A, B, LeftChild, RightChild],
        InterLevel is Level + 1,
        find_candidate(Higher, RightChild, NewRightChild, Candidate,
                       InterLevel, NewLevel),
        NewTree =.. [Type, A, B, LeftChild, NewRightChild].
```

B.1.4 check.pl

```
:- module(check, [check/6, split_bundle/4]).
:- use_module(unify).

% This is the overall algorithm for checking.  We begin with a special
% exception for unchecked adjuncts, since all they require is evidence that
% there was some point at which a compatible nonadjunct entered the tree.
check(Unchecked1, Unchecked2, Checked1, Checked2, Direction1, Direction2) :-
        last_entity_simple(Unchecked2, Adjunct),
        split_bundle(Adjunct, _, Collection2, _),
        collect_adjunction(Collection2, Unchecked1, Collection1),
        fix_project(Unchecked1, Collection2, Checked1, Unchecked2,
                    Direction1, Direction2),
        fix_child(Unchecked2, Collection1, Checked2, Unchecked1,
                  Direction2, Direction1).
check(Unchecked1, Unchecked2, Checked1, Checked2, Direction1, Direction2) :-
        collect(Unchecked1, Collection1),
        collect(Unchecked2, Collection2),
        unify(Collection1, Collection2, _),
```

```
        fix_project(Unchecked1, Collection2, Checked1, Unchecked2,
                    Direction1, Direction2),
        fix_child(Unchecked2, Collection1, Checked2, Unchecked1,
                  Direction2, Direction1).

% This collects features along the path for unification assuming that they
% the path has collectable features.
collect(A=>B, List) :-
        split_bundle(A, Type, Features, _),
        member(Type, [ch, chp, 'CH']), !,
        delete(Features, _:0, FixedFeatures),
        % The above means that singletons shouldn't propagate.
        % ie, things released by passive agents shouldn't also require
        % passive agents.
        collect(B, AnotherList),
        % We need to let features in B override features in A.
        findall(X:Y, (member(X:Y, FixedFeatures), member(X:_, AnotherList)),
                Overrides),
        override(Overrides, FixedFeatures, MoreFixedFeatures),
        append(MoreFixedFeatures, AnotherList, List).
collect(B=>_, List) :-
        split_bundle(B, Type, List, _),
        member(Type, [unch, unchp, 'UNCH']).
collect(B, List) :-
        split_bundle(B, _, List, _).

% This performs the override mentioned above.  ie, it just deletes.
override([], X, X).
override([A | Rest], X, Y) :-
        delete(X, A, Z),
        override(Rest, Z, Y).

% This collects for the adjunction case, but it only looks for a
% single compatble feature.
collect_adjunction(List1, A->_, List2) :-
        split_bundle(A, Type, SomeList2, _),
        member(Type, ['CH', ch, unch]),
        delete(SomeList2, _:0, List2),
        unify(List1, List2, _).
collect_adjunction(List1, A, List2) :-
        split_bundle(A, Type, SomeList2, _),
        member(Type, ['CH', ch, unch]),
        delete(SomeList2, _:0, List2),
        unify(List1, List2, _).
collect_adjunction(List1, A=>B, List2) :-
        split_bundle(A, Type, _, _),
        member(Type, ['UNCH', unchp]),
        collect_adjunction(List1, B, List2).
```

```
% This splits up a feature bundle.
split_bundle(Bundle:Direction, Type, Features, Direction) :-
        memberchk(Direction, [left, right]),
        Bundle =.. [Type | Features].
split_bundle(Bundle, Type, Features, both) :-
        Bundle =.. [Type | Features].

% These predicates actually do the checking given the things to be
% checked and the things they contain.

fix_project(A=>B, Collection, C=>D, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(AStatus, OtherStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection),
        D = B.
fix_project(A=>B, Collection, C=>D, Other, OneDirection, OtherDirection) :-
        split_bundle(A, AStatus, _, _),
        member(AStatus, [ch, 'CH', chp]),
        C = A,
        fix_project(B, Collection, D, Other, OneDirection, OtherDirection).
fix_project(A, Collection, C, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(AStatus, OtherStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection).

% Splitting them up by parent vs. child is an easy way to distinguish
% between who projected and who didn't.

fix_child(A=>B, Collection, C=>D, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity_child(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(OtherStatus, AStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
```

```
            ;   [AStatus, OtherStatus] = ['UNCH', unch])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection),
        D = B.
fix_child(A=>B, Collection, C=>D, Other, OneDirection, OtherDirection) :-
        split_bundle(A, AStatus, _, _),
        member(AStatus, [ch, 'CH', chp]),
        C = A,
        fix_child(B, Collection, D, Other, OneDirection, OtherDirection).
fix_child(A, Collection, C, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity_child(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(OtherStatus, AStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection).

% Finds the last bundle compatible to a counter part on the feature path.
last_entity(C=>_, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, SomeFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, Features, _),
        (   Mine = unchp
        ->  delete(SomeFeatures, _:0, TheirFeatures)
        ;   TheirFeatures = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity(C, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, SomeFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, Features, _),
        (   Mine = unchp
        ->  delete(SomeFeatures, _:0, TheirFeatures)
        ;   TheirFeatures = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity(_=>B, Counterpart, C) :-
        last_entity(B, Counterpart, C).


last_entity_child(C=>_, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, TheirFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, SomeFeatures, _),
        (   Theirs = unchp
        ->  delete(SomeFeatures, _:0, Features)
```

```
        ;   Features = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity_child(C, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, TheirFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, SomeFeatures, _),
        (   Theirs = unchp
        ->  delete(SomeFeatures, _:0, Features)
        ;   Features = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity_child(_->B, Counterpart, C) :-
        last_entity(B, Counterpart, C).

% this looks for unchecked adjuncts.
last_entity_simple(C->_, C) :-
        split_bundle(C, unchp, _, _).
last_entity_simple(C, C) :-
        split_bundle(C, unchp, _, _).
last_entity_simple(A->B, C) :-
        split_bundle(A, Type, _, _),
        member(Type, [ch, 'CH', chp]),
        last_entity_simple(B, C).

% check(Before, After, With) -- in the presence of With, Before becomes
% After.
check(unch, ch, 'UNCH').
check(unchp, unchp, 'CH').
check(unchp, chp, unch).
check(unchp, chp, ch).
check('UNCH', 'CH', unch).
check('CH', 'CH', unchp).
check(unch, unch, unchp).
check(ch, ch, unchp).

% project(Projecting, Over) -- Projecting projects over Over.
project('UNCH', unch).
project('ch', unchp).
project('unch', unchp).
project('CH', unchp).
```

B.1.5 unify.pl

```
:- module(unify, [unify/3]).

% This performs feature set unification on lists of features of the
% form P:Q. It stipulates that there must be at least one feature in
% common between the two of them before unification can take place, so,
% for instance, empty lists are not unifiable.
% Also, please note that if one feature set has duplicate P:Q pairs,
% then there wll be duplicates in the result.
```

```
unify(A, B, D) :- unifiable(A, B),
        check_singletons(A, B),
        check_singletons(B, A),
        do_unify(A, B, C), do_unify(B, C, D).

unifiable(A, B) :- unzip(A, X), unzip(B, Y), member(One, X), member(One, Y).

unzip([], []).
unzip([P:_ | Rest], [P | OtherRest]) :- unzip(Rest, OtherRest).

do_unify([], Other, Other) :- !.
do_unify([P:Q | Rest], Other, [P:Q | ResultRest]) :-
        member(P:R, Other), R = Q, !,
        delete(Other, P:Q, SmallerOther),
        do_unify(Rest, SmallerOther, ResultRest).
do_unify([P:Q | _], Other, _) :- member(P:R, Other), R \= Q, !, fail.
do_unify([P:Q | Rest], Other, [P:Q | ResultRest]) :-
        do_unify(Rest, Other, ResultRest).

% We check for things like by:0.  If such a thing is present in one, it MUST
% be present in the other.
check_singletons(A, B) :-
        findall(X:0, member(X:0, A), ASingletons),
        forall(member(Z, ASingletons), member(Z, B)).
```

B.1.6 portrayal.pl

```
:- use_module(check).

% This file adds a portray/1 handler for the system to print out
% correct terms.  All of this code is just display.

% This one is for trees.
portray(maxnode(A, B, C, D)) :- tree_display(maxnode(A, B, C, D), 0).

tree_display(maxnode(A, B, none, none), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('['),
        write(A),
        features_display(B),
        write(']').
tree_display(maxnode(A, B, LeftChild, RightChild), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('(['),
        write(A),
        features_display(B),
```

```
        write(']'),
        M is N + 1,
        tree_display(LeftChild, M),
        tree_display(RightChild, M),
        write(')').
tree_display(node(A, _, none, none), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write(A).
tree_display(node(A, _, LeftChild, RightChild), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('('),
        print(A),
        M is N + 1,
        tree_display(LeftChild, M),
        tree_display(RightChild, M),
        write(')').
tree_display(trace(A), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('<'),
        write(A),
        write('>').
tree_display(A, N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write(A).

features_display([]) :- !.
features_display([A]) :-
        tab(1), feature_display(A), !.
features_display([A | Rest]) :-
        tab(1), write('{'),
        list_features_display([A | Rest], first),
        write('}').

list_features_display([], _).
list_features_display([A], first) :-
        feature_display(A).
list_features_display([A], after) :-
        write(', '), feature_display(A).
list_features_display([A | Rest], first) :-
        feature_display(A),
        list_features_display(Rest, after).
```

```
list_features_display([A | Rest], after) :-
        write(',␣'),
        feature_display(A),
        list_features_display(Rest, after).


feature_display(Feature) :-
        split_bundle(Feature, unchp, Stuff, Direction),
        split_bundle(PrintFeature, 'unch+', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature) :-
        split_bundle(Feature, chp, Stuff, Direction),
        split_bundle(PrintFeature, 'ch+', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature1=>Feature2) :-
        feature_display(Feature1),
        write('=>'),
        feature_display(Feature2).
feature_display(Feature) :-
        write(Feature).

% This portrayal is for entire derivations.
portray(deriv(A, B)) :-
        list_tree_display(A),
        (   B = []
        ->  write('␣|')
        ;   nl,
            write('|␣'),
            print_words(B)).

list_tree_display([]).
list_tree_display([A | Rest]) :-
        print(A),
        list_tree_display(Rest).

print_words([]).
print_words([A | Rest]) :-
        write(A),
        tab(1),
        print_words(Rest).
```

B.1.7 latin.lex.pl

```
lex(pater, [unch(case:nom, num:sg, gnd:masc)]).
lex(filium, [unch(case:acc, num:sg, gnd:masc)]).
lex(amat, ['UNCH'(case:nom, num:sg), 'UNCH'(case:acc)]).
lex(laetus, [unchp(case:nom, num:sg, gnd:masc)]).
lex(laetum, [unchp(case:acc, num:sg, gnd:masc)]).
lex(a, ['UNCH'(case:abl):right=>unch(by:0)]).
lex(filio, [unch(case:abl, num:sg, gnd:masc)]).
```

```
lex(laeto , [unchp(case:abl , num:sg , gnd:masc)]).
lex(amatur , ['UNCH'(case:nom, num:sg , gnd:masc),
              'UNCH'(by:0)]).
```

B.1.8 english.lex.pl

```
% This contains the lexical items for English used in the text.

lex(the , ['UNCH'(type:n)->unch(type:n)]).
lex(happy , [unchp(type:n):right]).
lex(father , [unch(type:n, num:sg)]).
lex(son , [unch(type:n, num:sg)]).
lex(loves , ['UNCH'(type:n, num:sg):left , 'UNCH'(type:n):right]).
lex(is, ['UNCH'(type:n, num:sg):left , 'UNCH'(type:v):right]).
lex(loved , ['UNCH'(by:0):right->unch(type:v)]).
lex(by, ['UNCH'(type:n):right->unch(by:0)]).
```

B.2 The Output

We present several parses produced by the code given above. The first four parses are the examples from the third chapter in the order in which they appear: two Latin examples followed by two English examples. We follow them with four variations of those sentences.

pater laetus filium amat laetum:

```
?- parse([pater,laetus,filium,amat,laetum]).

Initial state:
| pater laetus filium amat laetum

Step 1:
[pater unch(case:nom, num:sg, gnd:masc)]
| laetus filium amat laetum

Step 2:
[pater unch(case:nom, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| filium amat laetum

Step 3:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
| filium amat laetum

Step 4:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
[filium unch(case:acc, num:sg, gnd:masc)]
| amat laetum

Step 5:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
[filium unch(case:acc, num:sg, gnd:masc)]
[amat {UNCH(case:nom, num:sg), UNCH(case:acc)}]
| laetum

Step 6:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([amat {CH(case:acc, num:sg, gnd:masc), UNCH(case:nom, num:sg)}]
  [filium ch(case:acc, num:sg, gnd:masc)]
  amat)
```

| laetum

Step 7:
([amat {CH(case:nom, num:sg, gnd:masc), CH(case:acc, num:sg, gnd:masc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  (amat
    [filium ch(case:acc, num:sg, gnd:masc)]
    amat))
| laetum

Step 8:
([amat {CH(case:nom, num:sg, gnd:masc), CH(case:acc, num:sg, gnd:masc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  (amat
    [filium ch(case:acc, num:sg, gnd:masc)]
    amat))
[laetum unch+(case:acc, num:sg, gnd:masc)] |

Step 9:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    (amat
      [filium ch(case:acc, num:sg, gnd:masc)]
      amat))
  [laetum unch+(case:acc, num:sg, gnd:masc)]) |

Step 10:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    (amat
      <filium>
      amat))
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)])) |

That was the last possible step.

pater laetus a filio laeto amatur:

```
?- parse([pater,laetus,a,filio,laeto,amatur]).

Initial state:
| pater laetus a filio laeto amatur

Step 1:
[pater unch(case:nom, num:sg, gnd:masc)]
| laetus a filio laeto amatur

Step 2:
[pater unch(case:nom, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| a filio laeto amatur

Step 3:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
| a filio laeto amatur

Step 4:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
[a UNCH(case:abl):right->unch(by:0)]
| filio laeto amatur

Step 5:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
[a UNCH(case:abl):right->unch(by:0)]
[filio unch(case:abl, num:sg, gnd:masc)]
| laeto amatur

Step 6:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
| laeto amatur

Step 7:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
```

```
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
[laeto unch+(case:abl, num:sg, gnd:masc)]
| amatur

Step 8:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    [filio ch(case:abl, num:sg, gnd:masc)])
  [laeto unch+(case:abl, num:sg, gnd:masc)])
| amatur

Step 9:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
| amatur

Step 10:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
[amatur UNCH(case:nom, num:sg, gnd:masc), UNCH(by:0)] |

Step 11:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
([amatur CH(by:0, case:abl, num:sg, gnd:masc), UNCH(case:nom, num:sg, gnd:masc)]
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    (a
```

```
      a
      <filio>)
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  amatur) |

Step 12:
([amatur CH(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      (a
        a
        <filio>)
      ([filio ch(case:abl, num:sg, gnd:masc)]
        filio
        [laeto ch+(case:abl, num:sg, gnd:masc)]))
    amatur)) |
```

That was the last possible step.

the happy father loves the happy son:

```
?- parse([the,happy,father,loves,the,happy,son]).

Initial state:
| the happy father loves the happy son

Step 1:
[the UNCH(type:n)->unch(type:n)]
| happy father loves the happy son

Step 2:
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
| father loves the happy son

Step 3:
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
[father unch(type:n, num:sg)]
| loves the happy son

Step 4:
[the UNCH(type:n)->unch(type:n)]
([father unch(type:n, num:sg)]
  [happy ch+(type:n):right]
```

```
  father)
| loves the happy son

Step 5:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  ([father ch(type:n, num:sg)]
    [happy ch+(type:n):right]
    father))
| loves the happy son

Step 6:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  ([father ch(type:n, num:sg)]
    [happy ch+(type:n):right]
    father))
[loves {UNCH(type:n, num:sg):left, UNCH(type:n):right}]
| the happy son

Step 7:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
| the happy son

Step 8:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
[the UNCH(type:n)->unch(type:n)]
| happy son

Step 9:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
```

```
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
| son

Step 10:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
[son unch(type:n, num:sg)] |

Step 11:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
[the UNCH(type:n)->unch(type:n)]
([son unch(type:n, num:sg)]
  [happy ch+(type:n):right]
  son) |

Step 12:
([loves {CH(type:n, num:sg):left, UNCH(type:n):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    ([father ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      father))
  loves)
([the CH(type:n, num:sg)->unch(type:n)]
  the
  ([son ch(type:n, num:sg)]
    [happy ch+(type:n):right]
    son)) |

Step 13:
([loves {CH(type:n, num:sg):right, CH(type:n, num:sg):left}]
  (loves
    ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
      the
      ([father ch(type:n, num:sg)]
```

```
        [happy ch+(type:n):right]
        father))
    loves)
  ([the CH(type:n, num:sg)->ch(type:n)]
    the
    ([son ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      son))) |
```

That was the last possible step.

the son is loved by the father:

```
?- parse([the,son,is,loved,by,the,father]).

Initial state:
| the son is loved by the father

Step 1:
[the UNCH(type:n)->unch(type:n)]
| son is loved by the father

Step 2:
[the UNCH(type:n)->unch(type:n)]
[son unch(type:n, num:sg)]
| is loved by the father

Step 3:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  [son ch(type:n, num:sg)])
| is loved by the father

Step 4:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  [son ch(type:n, num:sg)])
[is {UNCH(type:n, num:sg):left, UNCH(type:v):right}]
| loved by the father

Step 5:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
| loved by the father

Step 6:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
```

```
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
| by the father

Step 7:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
| the father

Step 8:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
| father

Step 9:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
[father unch(type:n, num:sg)] |

Step 10:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
([the CH(type:n, num:sg)->unch(type:n)]
  the
```

```
  [father ch(type:n, num:sg)]) |

Step 11:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
([by CH(type:n, num:sg):right->unch(by:0)]
  by
  ([the CH(type:n, num:sg)->ch(type:n)]
    the
    [father ch(type:n, num:sg)])) |

Step 12:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [son ch(type:n, num:sg)])
  is)
([loved CH(by:0, type:n, num:sg):right->unch(type:v)]
  loved
  ([by CH(type:n, num:sg):right->ch(by:0)]
    by
    ([the CH(type:n, num:sg)->ch(type:n)]
      the
      [father ch(type:n, num:sg)]))) |

Step 13:
([is {CH(type:v, num:sg):right, CH(type:n, num:sg):left}]
  (is
    ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
      the
      [son ch(type:n, num:sg)])
    is)
  ([loved CH(by:0, type:n, num:sg):right->ch(type:v)]
    loved
    ([by CH(type:n, num:sg):right->ch(by:0)]
      by
      ([the CH(type:n, num:sg)->ch(type:n)]
        the
        [father ch(type:n, num:sg)])))) |
```

That was the last possible step.

pater laetus amat filium laetum:

```
?- parse([pater,laetus,amat,filium,laetum]).
```

```
Initial state:
| pater laetus amat filium laetum

Step 1:
[pater unch(case:nom, num:sg, gnd:masc)]
| laetus amat filium laetum

Step 2:
[pater unch(case:nom, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| amat filium laetum

Step 3:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
| amat filium laetum

Step 4:
([pater unch(case:nom, num:sg, gnd:masc)]
  pater
  [laetus ch+(case:nom, num:sg, gnd:masc)])
[amat {UNCH(case:nom, num:sg), UNCH(case:acc)}]
| filium laetum

Step 5:
([amat {CH(case:nom, num:sg, gnd:masc), UNCH(case:acc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  amat)
| filium laetum

Step 6:
([amat {CH(case:nom, num:sg, gnd:masc), UNCH(case:acc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  amat)
[filium unch(case:acc, num:sg, gnd:masc)]
| laetum

Step 7:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    amat)
```

```
  [filium ch(case:acc, num:sg, gnd:masc)])
| laetum

Step 8:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    amat)
  [filium ch(case:acc, num:sg, gnd:masc)])
[laetum unch+(case:acc, num:sg, gnd:masc)] |

Step 9:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    (amat
      ([pater ch(case:nom, num:sg, gnd:masc)]
        pater
        [laetus ch+(case:nom, num:sg, gnd:masc)])
      amat)
    [filium ch(case:acc, num:sg, gnd:masc)])
  [laetum unch+(case:acc, num:sg, gnd:masc)]) |

Step 10:
([amat {CH(case:acc, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amat
    (amat
      ([pater ch(case:nom, num:sg, gnd:masc)]
        pater
        [laetus ch+(case:nom, num:sg, gnd:masc)])
      amat)
    <filium>)
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)])) |
```

That was the last possible step.

pater laetus a filio amatur laeto:

```
?- parse([pater,laetus,a,filio,amatur,laeto]).

Initial state:
| pater laetus a filio amatur laeto

Step 1:
[pater unch(case:nom, num:sg, gnd:masc)]
| laetus a filio amatur laeto
```

```
Step 2:
[pater unch(case:nom, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| a filio amatur laeto

Step 3:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
| a filio amatur laeto

Step 4:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
[a UNCH(case:abl):right->unch(by:0)]
| filio amatur laeto

Step 5:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
[a UNCH(case:abl):right->unch(by:0)]
[filio unch(case:abl, num:sg, gnd:masc)]
| amatur laeto

Step 6:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
   a
   [filio ch(case:abl, num:sg, gnd:masc)])
| amatur laeto

Step 7:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
   a
   [filio ch(case:abl, num:sg, gnd:masc)])
[amatur {UNCH(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
| laeto

Step 8:
([pater unch(case:nom, num:sg, gnd:masc)]
   pater
   [laetus ch+(case:nom, num:sg, gnd:masc)])
```

```
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH(case:nom, num:sg, gnd:masc)}]
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    a
    [filio ch(case:abl, num:sg, gnd:masc)])
  amatur)
| laeto

Step 9:
([amatur {CH(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      [filio ch(case:abl, num:sg, gnd:masc)])
    amatur))
| laeto

Step 10:
([amatur {CH(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      [filio ch(case:abl, num:sg, gnd:masc)])
    amatur))
[laeto unch+(case:abl, num:sg, gnd:masc)] |

Step 11:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amatur
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    (amatur
      ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
        a
        [filio ch(case:abl, num:sg, gnd:masc)])
      amatur))
  [laeto unch+(case:abl, num:sg, gnd:masc)]) |

Step 12:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  (amatur
    ([pater ch(case:nom, num:sg, gnd:masc)]
      pater
```

```
      [laetus ch+(case:nom, num:sg, gnd:masc)])
    (amatur
      ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
        a
        <filio>)
      amatur))
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)])) |
```

That was the last possible step.

a filio laeto pater amatur laetus:

```
?- parse([a,filio,laeto,pater,amatur,laetus]).

Initial state:
| a filio laeto pater amatur laetus

Step 1:
[a UNCH(case:abl):right->unch(by:0)]
| filio laeto pater amatur laetus

Step 2:
[a UNCH(case:abl):right->unch(by:0)]
[filio unch(case:abl, num:sg, gnd:masc)]
| laeto pater amatur laetus

Step 3:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
| laeto pater amatur laetus

Step 4:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
[laeto unch+(case:abl, num:sg, gnd:masc)]
| pater amatur laetus

Step 5:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    [filio ch(case:abl, num:sg, gnd:masc)])
  [laeto unch+(case:abl, num:sg, gnd:masc)])
| pater amatur laetus

Step 6:
```

```
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
| pater amatur laetus

Step 7:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
[pater unch(case:nom, num:sg, gnd:masc)]
| amatur laetus

Step 8:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
[pater unch(case:nom, num:sg, gnd:masc)]
[amatur {UNCH(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
| laetus

Step 9:
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  (a
    a
    <filio>)
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
([amatur {CH(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
  [pater ch(case:nom, num:sg, gnd:masc)]
  amatur)
| laetus

Step 10:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    (a
```

```
      a
      <filio>)
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  (amatur
    [pater ch(case:nom, num:sg, gnd:masc)]
    amatur))
| laetus

Step 11:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), CH(case:nom, num:sg, gnd:masc)}]
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    (a
      a
      <filio>)
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  (amatur
    [pater ch(case:nom, num:sg, gnd:masc)]
    amatur))
[laetus unch+(case:nom, num:sg, gnd:masc)] |

Step 12:
([amatur {CH(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      (a
        a
        <filio>)
      ([filio ch(case:abl, num:sg, gnd:masc)]
        filio
        [laeto ch+(case:abl, num:sg, gnd:masc)]))
    (amatur
      [pater ch(case:nom, num:sg, gnd:masc)]
      amatur))
  [laetus unch+(case:nom, num:sg, gnd:masc)]) |

Step 13:
([amatur {CH(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      (a
        a
        <filio>)
      ([filio ch(case:abl, num:sg, gnd:masc)]
        filio
        [laeto ch+(case:abl, num:sg, gnd:masc)]))
```

```
    (amatur
      <pater>
      amatur))
  ([pater ch(case:nom, num:sg, gnd:masc)]
    pater
    [laetus ch+(case:nom, num:sg, gnd:masc)])) |
```

That was the last possible step.

the father is loved by the happy son:

```
?- parse([the,father,is,loved,by,the,happy,son]).

Initial state:
| the father is loved by the happy son

Step 1:
[the UNCH(type:n)->unch(type:n)]
| father is loved by the happy son

Step 2:
[the UNCH(type:n)->unch(type:n)]
[father unch(type:n, num:sg)]
| is loved by the happy son

Step 3:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  [father ch(type:n, num:sg)])
| is loved by the happy son

Step 4:
([the CH(type:n, num:sg)->unch(type:n)]
  the
  [father ch(type:n, num:sg)])
[is {UNCH(type:n, num:sg):left, UNCH(type:v):right}]
| loved by the happy son

Step 5:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
| loved by the happy son

Step 6:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
```

```
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
| by the happy son

Step 7:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
| the happy son

Step 8:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
| happy son

Step 9:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
| son

Step 10:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
[happy unch+(type:n):right]
[son unch(type:n, num:sg)] |
```

```
Step 11:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
[the UNCH(type:n)->unch(type:n)]
([son unch(type:n, num:sg)]
  [happy ch+(type:n):right]
  son) |

Step 12:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
[by UNCH(type:n):right->unch(by:0)]
([the CH(type:n, num:sg)->unch(type:n)]
  the
  ([son ch(type:n, num:sg)]
    [happy ch+(type:n):right]
    son)) |

Step 13:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
[loved UNCH(by:0):right->unch(type:v)]
([by CH(type:n, num:sg):right->unch(by:0)]
  by
  ([the CH(type:n, num:sg)->ch(type:n)]
    the
    ([son ch(type:n, num:sg)]
      [happy ch+(type:n):right]
      son))) |

Step 14:
([is {CH(type:n, num:sg):left, UNCH(type:v):right}]
  ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
    the
    [father ch(type:n, num:sg)])
  is)
```

```
([loved CH(by:0, type:n, num:sg):right->unch(type:v)]
  loved
  ([by CH(type:n, num:sg):right->ch(by:0)]
    by
    ([the CH(type:n, num:sg)->ch(type:n)]
      the
      ([son ch(type:n, num:sg)]
        [happy ch+(type:n):right]
        son)))) |
```

Step 15:

```
([is {CH(type:v, num:sg):right, CH(type:n, num:sg):left}]
  (is
    ([the CH(type:n, num:sg)->ch(type:n, num:sg)]
      the
      [father ch(type:n, num:sg)])
    is)
  ([loved CH(by:0, type:n, num:sg):right->ch(type:v)]
    loved
    ([by CH(type:n, num:sg):right->ch(by:0)]
      by
      ([the CH(type:n, num:sg)->ch(type:n)]
        the
        ([son ch(type:n, num:sg)]
          [happy ch+(type:n):right]
          son))))) |
```

That was the last possible step.

Appendix C

The Program—Chapter 4 version

C.1 Source Code

Much of the code for the version of the parser that handles optional subjects is the same as the code that does not. We therefore present only the files that were changed to support the functionality mentioned in the fourth chapter. Even so, they are very similar to the files of the other version.

C.1.1 parser.pl

```
% This file is the main loop of the program.  It goes through
% each step and prints it out, numbered.
% You have to load a lexicon file before you use this.

:- use_module(deriv).
:- use_module(check).
:- [portrayal].

parse(A) :-
        do_parse(deriv([], A), 0).

% Every call to step is cut so that only the first solution is taken.
do_parse(Deriv, 0) :-
        nl, write('Initial state:'),
        print(Deriv),
        step(Deriv, NextDeriv), !,
        do_parse(NextDeriv, 1).
do_parse(Deriv, N) :-
        N > 0,
        nl, nl, write('Step '), write(N), write(':'),
        print(Deriv),
        step(Deriv, NextDeriv), !,
        M is N + 1,
        do_parse(NextDeriv, M).
do_parse(A, _) :-
        nl, nl, write('Checking any unchecked optional features:'),
```

```
        optifix(A, Fixed), !,
        print(Fixed), nl, nl,
        write('That was the last possible step.').

% All of this below checks unchecked optional features in the derivation by
% recursing down the trees.
optifix(deriv(Trees, Rest), deriv(FixedTrees, Rest)) :-
        optifix_trees(Trees, FixedTrees).

optifix_trees([], []).
optifix_trees([Tree | Rest], [FixedTree | FixedRest]) :-
        optifix_tree(Tree, FixedTree),
        optifix_trees(Rest, FixedRest).

optifix_tree(maxnode(A, Features, Left, Right),
             maxnode(A, FixedFeatures, FixedLeft, FixedRight)) :-
        optifix_features(Features, FixedFeatures),
        optifix_tree(Left, FixedLeft),
        optifix_tree(Right, FixedRight).
optifix_tree(node(A, Features, Left, Right),
             node(A, FixedFeatures, FixedLeft, FixedRight)) :-
        optifix_features(Features, FixedFeatures),
        optifix_tree(Left, FixedLeft),
        optifix_tree(Right, FixedRight).
optifix_tree(A, A).

optifix_features([], []).
optifix_features([Feature | Rest], [FixedFeature | FixedRest]) :-
        optifix_feature(Feature, FixedFeature),
        optifix_features(Rest, FixedRest).

optifix_feature(Feature, FixedFeature) :-
        split_bundle(Feature, 'UNCH?', A, B),
        split_bundle(FixedFeature, 'CH?', A, B).
optifix_feature(Feature, FixedFeature) :-
        split_bundle(Feature, 'unchp?', A, B),
        split_bundle(FixedFeature, 'chp?', A, B).
optifix_feature(A, A).
```

C.1.2 check.pl

```
:- module(check, [check/6, split_bundle/4]).
:- use_module(unify).

% This is the overall algorithm for checking.  We begin with a special
% exception for unchecked adjuncts, since all they require is evidence that
% there was some point at which a compatible nonadjunct entered the tree.
check(Unchecked1, Unchecked2, Checked1, Checked2, Direction1, Direction2) :-
        last_entity_simple(Unchecked2, Adjunct),
        split_bundle(Adjunct, _, Collection2, _),
```

```
        collect_adjunction(Collection2, Unchecked1, Collection1),
        fix_project(Unchecked1, Collection2, Checked1, Unchecked2,
                    Direction1, Direction2),
        fix_child(Unchecked2, Collection1, Checked2, Unchecked1,
                  Direction2, Direction1).
check(Unchecked1, Unchecked2, Checked1, Checked2, Direction1, Direction2) :-
        collect(Unchecked1, Collection1),
        collect(Unchecked2, Collection2),
        unify(Collection1, Collection2, _),
        fix_project(Unchecked1, Collection2, Checked1, Unchecked2,
                    Direction1, Direction2),
        fix_child(Unchecked2, Collection1, Checked2, Unchecked1,
                  Direction2, Direction1).

% This collects features along the path for unification assuming that
% the path has collectable features.
collect(A=>B, List) :-
        split_bundle(A, Type, Features, _),
        member(Type, [ch, chp, 'chp?', 'CH', 'CH?']), !,
        delete(Features, _:0, FixedFeatures),
        % The above means that singletons shouldn't propagate.
        % ie, things released by passive agents shouldn't also require
        % passive agents.
        collect(B, AnotherList),
        % We need to let features in B override features in A.
        findall(X:Y, (member(X:Y, FixedFeatures), member(X:_, AnotherList)),
                Overrides),
        override(Overrides, FixedFeatures, MoreFixedFeatures),
        append(MoreFixedFeatures, AnotherList, List).
collect(B=>_, List) :-
        split_bundle(B, Type, List, _),
        member(Type, [unch, unchp, 'unchp?', 'UNCH', 'UNCH?']).
collect(B, List) :-
        split_bundle(B, _, List, _).

% This performs the override mentioned above.  ie, it just deletes.
override([], X, X).
override([A | Rest], X, Y) :-
        delete(X, A, Z),
        override(Rest, Z, Y).

% This collects for the adjunction case, but it only looks for a
% single compatble feature.
collect_adjunction(List1, A=>_, List2) :-
        split_bundle(A, Type, SomeList2, _),
        member(Type, ['CH', 'CH?', 'UNCH?', ch, unch]),
        delete(SomeList2, _:0, List2),
        unify(List1, List2, _).
collect_adjunction(List1, A, List2) :-
```

```
        split_bundle(A, Type, SomeList2, _),
        member(Type, ['CH', 'CH?', 'UNCH?', ch, unch]),
        delete(SomeList2, _:0, List2),
        unify(List1, List2, _).
collect_adjunction(List1, A=>B, List2) :-
        split_bundle(A, Type, _, _),
        member(Type, ['UNCH', unchp, 'unchp?']),
        collect_adjunction(List1, B, List2).

% This splits up a feature bundle.
split_bundle(Bundle:Direction, Type, Features, Direction) :-
        memberchk(Direction, [left, right]),
        Bundle =.. [Type | Features].
split_bundle(Bundle, Type, Features, both) :-
        Bundle =.. [Type | Features].

% These predicates actually do the checking given the things to be
% checked and the things they contain.

fix_project(A=>B, Collection, C=>D, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(AStatus, OtherStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch]
            ;   [AStatus, OtherStatus] = [unch, 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', unch]
            ;   [AStatus, OtherStatus] = ['unchp', 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', 'unchp'])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection),
        D = B.
fix_project(A=>B, Collection, C=>D, Other, OneDirection, OtherDirection) :-
        split_bundle(A, AStatus, _, _),
        member(AStatus, [ch, 'CH', chp]),
        C = A,
        fix_project(B, Collection, D, Other, OneDirection, OtherDirection).
fix_project(A, Collection, C, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(AStatus, OtherStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch]
            ;   [AStatus, OtherStatus] = [unch, 'UNCH?']
```

```
            ;   [AStatus, OtherStatus] = ['UNCH?', unch]
            ;   [AStatus, OtherStatus] = ['unchp', 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', 'unchp'])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection).

% Splitting them up by parent vs. child is an easy way to distinguish
% between who projected and who didn't.

fix_child(A->>B, Collection, C->>D, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity_child(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(OtherStatus, AStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch]
            ;   [AStatus, OtherStatus] = [unch, 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', unch]
            ;   [AStatus, OtherStatus] = ['unchp', 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', 'unchp'])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection),
        D = B.
fix_child(A->>B, Collection, C->>D, Other, OneDirection, OtherDirection) :-
        split_bundle(A, AStatus, _, _),
        member(AStatus, [ch, 'CH', chp]),
        C = A,
        fix_child(B, Collection, D, Other, OneDirection, OtherDirection).
fix_child(A, Collection, C, Other, ADirection, OtherDirection) :-
        split_bundle(A, AStatus, AFeatures, ADirection),
        last_entity_child(Other, A, LastOther),
        split_bundle(LastOther, OtherStatus, _, OtherDirection),
        project(OtherStatus, AStatus),
        check(AStatus, NewAStatus, OtherStatus),
        (   ([AStatus, OtherStatus] = [unch, 'UNCH']
            ;   [AStatus, OtherStatus] = ['UNCH', unch]
            ;   [AStatus, OtherStatus] = [unch, 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', unch]
            ;   [AStatus, OtherStatus] = ['unchp', 'UNCH?']
            ;   [AStatus, OtherStatus] = ['UNCH?', 'unchp'])
        ->  unify(Collection, AFeatures, NewAFeatures)
        ;   NewAFeatures = AFeatures),
        split_bundle(C, NewAStatus, NewAFeatures, ADirection).

% Finds the last bundle compatible to a counter part on the feature path.
last_entity(C->>_, Counterpart, C) :-
```

```
        split_bundle(Counterpart, Theirs, SomeFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, Features, _),
        (   Mine = unchp
        ->  delete(SomeFeatures, _:0, TheirFeatures)
        ;   TheirFeatures = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity(C, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, SomeFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, Features, _),
        (   Mine = unchp
        ->  delete(SomeFeatures, _:0, TheirFeatures)
        ;   TheirFeatures = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity(_->B, Counterpart, C) :-
        last_entity(B, Counterpart, C).


last_entity_child(C->_, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, TheirFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, SomeFeatures, _),
        (   Theirs = unchp
        ->  delete(SomeFeatures, _:0, Features)
        ;   Features = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity_child(C, Counterpart, C) :-
        split_bundle(Counterpart, Theirs, TheirFeatures, _),
        check(Theirs, _, Mine),
        split_bundle(C, Mine, SomeFeatures, _),
        (   Theirs = unchp
        ->  delete(SomeFeatures, _:0, Features)
        ;   Features = SomeFeatures),
        unify(TheirFeatures, Features, _).
last_entity_child(_->B, Counterpart, C) :-
        last_entity(B, Counterpart, C).

% this looks for unchecked adjuncts.
last_entity_simple(C->_, C) :-
        split_bundle(C, Unchecked, _, _),
        member(Unchecked, [unchp, 'unchp?']).
last_entity_simple(C, C) :-
        split_bundle(C, Unchecked, _, _),
        member(Unchecked, [unchp, 'unchp?']).
last_entity_simple(A->B, C) :-
        split_bundle(A, Type, _, _),
        member(Type, [ch, 'CH', chp, 'chp?']),
        last_entity_simple(B, C).
```

```
% check(Before, After, With) -- in the presence of With, Before becomes
% After.
check('unchp?', 'chp?', B) :- check(unchp, chp, B).
check('unchp?', 'unchp?', B) :- check(unchp, unchp, B).
check(unchp,unchp,'CH').
check(A, 'unchp?', B) :- check(A, unchp, B), B \= 'UNCH?'.
check(A, B, 'unchp?') :- check(A, B, unchp).
check(unch, ch, 'UNCH').
check(unch, ch, 'UNCH?').
check(unchp, unchp, 'CH?').
check(unchp, 'unchp?', 'UNCH?').
check(unchp, chp, unch).
check(unchp, chp, ch).
check('UNCH', 'CH', unch).
check('UNCH?', 'CH?', unch).
check('CH', 'CH', unchp).
check('CH?', 'CH?', unchp).
check('UNCH?', 'UNCH?', unchp).
check(unch, unch, unchp).
check(ch, ch, unchp).

% project(Projecting, Over) -- Projecting projects over Over.
project('unchp?', A) :- project(unchp, A).
project(A, 'unchp?') :- project(A, unchp), A \= 'UNCH?'.
project('UNCH', unch).
project('UNCH?', unch).
project('ch', unchp).
project('unch', unchp).
project('CH', unchp).
project('CH?', unchp).
project('UNCH?', unchp).
```

C.1.3 portrayal.pl

```
:- use_module(check).

% This file adds a portray/1 handler for the system to print out
% correct terms.  All of this code is just display.

% This one is for trees.
portray(maxnode(A, B, C, D)) :- tree_display(maxnode(A, B, C, D), 0).

tree_display(maxnode(A, B, none, none), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('['),
        write(A),
        features_display(B),
```

```
        write(']').
tree_display(maxnode(A, B, LeftChild, RightChild), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('([').
        write(A),
        features_display(B),
        write(']'),
        M is N + 1,
        tree_display(LeftChild, M),
        tree_display(RightChild, M),
        write(')').
tree_display(node(A, _, none, none), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write(A).
tree_display(node(A, _, LeftChild, RightChild), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('('),
        print(A),
        M is N + 1,
        tree_display(LeftChild, M),
        tree_display(RightChild, M),
        write(')').
tree_display(trace(A), N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write('<'),
        write(A),
        write('>').
tree_display(A, N) :-
        Tabs is N * 3,
        nl,
        tab(Tabs),
        write(A).

features_display([]) :- !.
features_display([A]) :-
        tab(1), feature_display(A), !.
features_display([A | Rest]) :-
        tab(1), write('{'),
        list_features_display([A | Rest], first),
        write('}').
```

```
list_features_display([], _).
list_features_display([A], first) :-
        feature_display(A).
list_features_display([A], after) :-
        write(', '), feature_display(A).
list_features_display([A | Rest], first) :-
        feature_display(A),
        list_features_display(Rest, after).
list_features_display([A | Rest], after) :-
        write(', '),
        feature_display(A),
        list_features_display(Rest, after).


feature_display(Feature) :-
        split_bundle(Feature, unchp, Stuff, Direction),
        split_bundle(PrintFeature, 'unch+', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature) :-
        split_bundle(Feature, chp, Stuff, Direction),
        split_bundle(PrintFeature, 'ch+', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature) :-
        split_bundle(Feature, 'unchp?', Stuff, Direction),
        split_bundle(PrintFeature, 'unch+?', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature) :-
        split_bundle(Feature, 'chp?', Stuff, Direction),
        split_bundle(PrintFeature, 'ch+?', Stuff, Direction),
        write(PrintFeature).
feature_display(Feature1->Feature2) :-
        feature_display(Feature1),
        write('->'),
        feature_display(Feature2).
feature_display(Feature) :-
        write(Feature).

% This portrayal is for entire derivations.
portray(deriv(A, B)) :-
        list_tree_display(A),
        (   B = []
        ->  write('_|')
        ;   nl,
            write('|_'),
            print_words(B)).

list_tree_display([]).
list_tree_display([A | Rest]) :-
        print(A),
```

```
        list_tree_display(Rest).

print_words([]).
print_words([A | Rest]) :-
        write(A),
        tab(1),
        print_words(Rest).
```

C.1.4 latin.lex.pl

```
lex(pater, [unch(case:nom, num:sg, gnd:masc)]).
lex(filium, [unch(case:acc, num:sg, gnd:masc)]).
lex(amat, ['UNCH?'(case:nom, num:sg), 'UNCH'(case:acc)]).
lex(laetus, [unchp(case:nom, num:sg, gnd:masc)]).
lex(laetum, [unchp(case:acc, num:sg, gnd:masc)]).
lex(a, ['UNCH'(case:abl):right=>unch(by:0)]).
lex(filio, [unch(case:abl, num:sg, gnd:masc)]).
lex(laeto, [unchp(case:abl, num:sg, gnd:masc)]).
lex(amatur, ['UNCH?'(case:nom, num:sg, gnd:masc),
             'UNCH'(by:0)]).
```

C.2 The Output

We provide the two parses given in Chapter 4 below as displayed by the program. Thereafter, we provide two additional variations in Latin.

filium laetus amat laetum:

```
?- parse([filium,laetus,amat,laetum]).

Initial state:
| filium laetus amat laetum

Step 1:
[filium unch(case:acc, num:sg, gnd:masc)]
| laetus amat laetum

Step 2:
[filium unch(case:acc, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| amat laetum

Step 3:
[filium unch(case:acc, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
[amat {UNCH?(case:nom, num:sg), UNCH(case:acc)}]
| laetum

Step 4:
[filium unch(case:acc, num:sg, gnd:masc)]
([amat {UNCH?(case:nom, num:sg, gnd:masc), UNCH(case:acc)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  amat)
| laetum

Step 5:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  [filium ch(case:acc, num:sg, gnd:masc)]
  (amat
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amat))
| laetum

Step 6:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  [filium ch(case:acc, num:sg, gnd:masc)]
  (amat
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amat))
[laetum unch+(case:acc, num:sg, gnd:masc)] |
```

```
Step 7:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amat
    [filium ch(case:acc, num:sg, gnd:masc)]
    (amat
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amat))
  [laetum unch+(case:acc, num:sg, gnd:masc)]) |

Step 8:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amat
    <filium>
    (amat
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amat))
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)])) |

Checking any unchecked optional features:
([amat {CH(case:acc, num:sg, gnd:masc), CH?(case:nom, num:sg, gnd:masc)}]
  (amat
    <filium>
    (amat
      [laetus ch+?(case:nom, num:sg, gnd:masc)]
      amat))
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)])) |

That was the last possible step.
```

filium laetus amat laetum pater:

```
?- parse([filium,laetus,amat,laetum,pater]).

Initial state:
| filium laetus amat laetum pater

Step 1:
[filium unch(case:acc, num:sg, gnd:masc)]
| laetus amat laetum pater

Step 2:
[filium unch(case:acc, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
| amat laetum pater

Step 3:
```

```
[filium unch(case:acc, num:sg, gnd:masc)]
[laetus unch+(case:nom, num:sg, gnd:masc)]
[amat {UNCH?(case:nom, num:sg), UNCH(case:acc)}]
| laetum pater

Step 4:
[filium unch(case:acc, num:sg, gnd:masc)]
([amat {UNCH?(case:nom, num:sg, gnd:masc), UNCH(case:acc)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  amat)
| laetum pater

Step 5:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  [filium ch(case:acc, num:sg, gnd:masc)]
  (amat
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amat))
| laetum pater

Step 6:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  [filium ch(case:acc, num:sg, gnd:masc)]
  (amat
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amat))
[laetum unch+(case:acc, num:sg, gnd:masc)]
| pater

Step 7:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amat
    [filium ch(case:acc, num:sg, gnd:masc)]
    (amat
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amat))
  [laetum unch+(case:acc, num:sg, gnd:masc)])
| pater

Step 8:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amat
    <filium>
    (amat
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amat))
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)]))
```

```
| pater

Step 9:
([amat {CH(case:acc, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amat
    <filium>
    (amat
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amat))
  ([filium ch(case:acc, num:sg, gnd:masc)]
    filium
    [laetum ch+(case:acc, num:sg, gnd:masc)]))
[pater unch(case:nom, num:sg, gnd:masc)] |

Step 10:
([amat {CH?(case:nom, num:sg, gnd:masc), CH(case:acc, num:sg, gnd:masc)}]
  (amat
    (amat
      <filium>
      (amat
        [laetus unch+?(case:nom, num:sg, gnd:masc)]
        amat))
    ([filium ch(case:acc, num:sg, gnd:masc)]
      filium
      [laetum ch+(case:acc, num:sg, gnd:masc)]))
  [pater ch(case:nom, num:sg, gnd:masc)]) |

Step 11:
([amat {CH?(case:nom, num:sg, gnd:masc), CH(case:acc, num:sg, gnd:masc)}]
  (amat
    (amat
      <filium>
      (amat
        <laetus>
        amat))
    ([filium ch(case:acc, num:sg, gnd:masc)]
      filium
      [laetum ch+(case:acc, num:sg, gnd:masc)]))
  ([pater ch(case:nom, num:sg, gnd:masc)]
    [laetus ch+?(case:nom, num:sg, gnd:masc)]
    pater)) |

Checking any unchecked optional features:
([amat {CH?(case:nom, num:sg, gnd:masc), CH(case:acc, num:sg, gnd:masc)}]
  (amat
    (amat
      <filium>
      (amat
        <laetus>
```

```
        amat))
    ([filium ch(case:acc, num:sg, gnd:masc)]
      filium
      [laetum ch+(case:acc, num:sg, gnd:masc)]))
  ([pater ch(case:nom, num:sg, gnd:masc)]
    [laetus ch+?(case:nom, num:sg, gnd:masc)]
    pater)) |
```

That was the last possible step.

laetus amatur a filio laeto:

```
?- parse([laetus,amatur,a,filio,laeto]).

Initial state:
| laetus amatur a filio laeto

Step 1:
[laetus unch+(case:nom, num:sg, gnd:masc)]
| amatur a filio laeto

Step 2:
[laetus unch+(case:nom, num:sg, gnd:masc)]
[amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
| a filio laeto

Step 3:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  amatur)
| a filio laeto

Step 4:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  amatur)
[a UNCH(case:abl):right->unch(by:0)]
| filio laeto

Step 5:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  amatur)
[a UNCH(case:abl):right->unch(by:0)]
[filio unch(case:abl, num:sg, gnd:masc)]
| laeto

Step 6:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
```

```
  amatur)
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
| laeto

Step 7:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amatur)
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    a
    [filio ch(case:abl, num:sg, gnd:masc)]))
| laeto

Step 8:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    amatur)
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    a
    [filio ch(case:abl, num:sg, gnd:masc)]))
[laeto unch+(case:abl, num:sg, gnd:masc)] |

Step 9:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    (amatur
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amatur)
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      [filio ch(case:abl, num:sg, gnd:masc)]))
  [laeto unch+(case:abl, num:sg, gnd:masc)]) |

Step 10:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    (amatur
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      amatur)
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      <filio>))
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)])) |
```

```
Checking any unchecked optional features:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), CH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    (amatur
      [laetus ch+?(case:nom, num:sg, gnd:masc)]
      amatur)
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      <filio>))
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)])) |
```

That was the last possible step.

laetus a filio amatur laeto pater:

```
?- parse([laetus,a,filio,amatur,laeto,pater]).

Initial state:
| laetus a filio amatur laeto pater

Step 1:
[laetus unch+(case:nom, num:sg, gnd:masc)]
| a filio amatur laeto pater

Step 2:
[laetus unch+(case:nom, num:sg, gnd:masc)]
[a UNCH(case:abl):right->unch(by:0)]
| filio amatur laeto pater

Step 3:
[laetus unch+(case:nom, num:sg, gnd:masc)]
[a UNCH(case:abl):right->unch(by:0)]
[filio unch(case:abl, num:sg, gnd:masc)]
| amatur laeto pater

Step 4:
[laetus unch+(case:nom, num:sg, gnd:masc)]
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
| amatur laeto pater

Step 5:
[laetus unch+(case:nom, num:sg, gnd:masc)]
([a CH(case:abl, num:sg, gnd:masc):right->unch(by:0)]
  a
  [filio ch(case:abl, num:sg, gnd:masc)])
```

```
[amatur {UNCH?(case:nom, num:sg, gnd:masc), UNCH(by:0)}]
| laeto pater

Step 6:
[laetus unch+(case:nom, num:sg, gnd:masc)]
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
    a
    [filio ch(case:abl, num:sg, gnd:masc)])
  amatur)
| laeto pater

Step 7:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      [filio ch(case:abl, num:sg, gnd:masc)])
    amatur))
| laeto pater

Step 8:
([amatur {UNCH?(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  [laetus unch+?(case:nom, num:sg, gnd:masc)]
  (amatur
    ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
      a
      [filio ch(case:abl, num:sg, gnd:masc)])
    amatur))
[laeto unch+(case:abl, num:sg, gnd:masc)]
| pater

Step 9:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    (amatur
      ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
        a
        [filio ch(case:abl, num:sg, gnd:masc)])
      amatur))
  [laeto unch+(case:abl, num:sg, gnd:masc)])
| pater

Step 10:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
```

```
    (amatur
      ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
        a
        <filio>)
      amatur))
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
| pater

Step 11:
([amatur {CH(by:0, case:abl, num:sg, gnd:masc), UNCH?(case:nom, num:sg, gnd:masc)}]
  (amatur
    [laetus unch+?(case:nom, num:sg, gnd:masc)]
    (amatur
      ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
        a
        <filio>)
      amatur))
  ([filio ch(case:abl, num:sg, gnd:masc)]
    filio
    [laeto ch+(case:abl, num:sg, gnd:masc)]))
[pater unch(case:nom, num:sg, gnd:masc)] |

Step 12:
([amatur {CH?(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  (amatur
    (amatur
      [laetus unch+?(case:nom, num:sg, gnd:masc)]
      (amatur
        ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
          a
          <filio>)
        amatur))
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  [pater ch(case:nom, num:sg, gnd:masc)]) |

Step 13:
([amatur {CH?(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  (amatur
    (amatur
      <laetus>
      (amatur
        ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
          a
          <filio>)
        amatur))
```

```
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  ([pater ch(case:nom, num:sg, gnd:masc)]
    [laetus ch+?(case:nom, num:sg, gnd:masc)]
    pater)) |
```

Checking any unchecked optional features:

```
([amatur {CH?(case:nom, num:sg, gnd:masc), CH(by:0, case:abl, num:sg, gnd:masc)}]
  (amatur
    (amatur
      <laetus>
      (amatur
        ([a CH(case:abl, num:sg, gnd:masc):right->ch(by:0)]
          a
          <filio>)
        amatur))
    ([filio ch(case:abl, num:sg, gnd:masc)]
      filio
      [laeto ch+(case:abl, num:sg, gnd:masc)]))
  ([pater ch(case:nom, num:sg, gnd:masc)]
    [laetus ch+?(case:nom, num:sg, gnd:masc)]
    pater)) |
```

That was the last possible step.