Reversibility in Natural Language Processing

Gertjan van Noord

$$
\texttt{?-}\mathit{sign}\left(\begin{bmatrix}
\mathit{phon} : \text{“negen neven negen”} \\
\mathit{syn} : \begin{bmatrix} \mathit{cat} : \text{s} \\ \mathit{subcat} : \langle\rangle \end{bmatrix} \\
\mathit{sem} : \begin{bmatrix}
\mathit{pred} : \text{bow} \\
\mathit{arg1} : \begin{bmatrix}
\mathit{quant} : \begin{bmatrix} \mathit{pred} : \text{nine} \end{bmatrix} \\
\mathit{arg} : \begin{bmatrix} \mathit{pred} : \text{cousin} \\ \mathit{number} : \text{pl} \end{bmatrix}
\end{bmatrix} \\
\mathit{tense} : \text{past}
\end{bmatrix}
\end{bmatrix}\right).
$$

Reversibility in Natural Language Processing

Omkeerbaarheid in natuurlijke-taalverwerking
(met een samenvatting in het Nederlands)

Proefschrift ter verkrijging van de graad van doctor
aan de Rijksuniversiteit te Utrecht
op gezag van Rector Magnificus, Prof.dr. J.A. van Ginkel
ingevolge het besluit van het College van Dekanen
in het openbaar te verdedigen
op 15 januari 1993 des namiddags te 14.30 uur

door
Gerardus Johannes Maria van Noord
geboren op 8 mei 1961, te Culemborg

promotoren:

Prof.dr. D.J.N. van Eijck, Faculteit der Letteren, Rijksuniversiteit Utrecht; Centrum voor Wiskunde en Informatica, Amsterdam.

Prof.ir. S.P.J. Landsbergen, Faculteit der Letteren, Rijksuniversiteit Utrecht; Instituut voor Perceptie Onderzoek, Eindhoven.

I was partially supported by the European Community and the NBBI through the Eurotra project, and the German Science Foundation in its Special Collaborative Research Programme on Artificial Intelligence and Knowledge Based Systems (SFB 314, Project N3 BiLD).

CIP-GEGEVENS KONINKLIJKE BIBLIOTHEEK, DEN HAAG

Noord, Gerardus Johannes Maria van

Reversibility in natural language processing / Gerardus
Johannes Maria van Noord. - [S.l. : s.n.]
Proefschrift Rijksuniversiteit Utrecht. - Met lit. opg. -
Met samenvatting in het Nederlands.
ISBN 90-9005661-0
Trefw.: natuurlijke taalverwerking.

Contents

Chapter 1

Introduction

Constraint-based grammars have become increasingly popular within the field of natural language processing. One of the reasons for this success is that constraint-based grammars are completely *declarative*; that is, the grammar only states facts of the language it describes, without stating how such a grammar should be used for parsing or generation. Such declarative grammars, for this reason, provide for an abstract level of language description, and are easy to understand, and hence relatively easy to debug, extend, and re-use in other applications.

Because constraint-based grammars do not enforce a specific processing regime, it is possible to conceive of constraint-based grammars, which can be used *both* for parsing and generation. Such grammars may be called *reversible* (Kay, 1975). Section 1.3 defines the notion *reversibility* somewhat more precisely, and discusses some of its properties. I define a reversible grammar as a grammar for which parsing and generation are both guaranteed to terminate. The notion of a grammar that can be used both for parsing and generation is intuitively appealing; but I provide explicit motivation for the use of reversible grammars in section 1.1.

Although declarative grammars can, in principle, be used in a reversible way, it turned out that from a practical point of view, several problems remained to be solved, in order for this ideal to be realized. Historically, constraint-based grammars often were used for parsing only. Attempts to use such grammars (written with parsing in mind) for generation, failed. Some of the problems are discussed in chapter 3. Specialized generation algorithms had to be developed, to be able to use constraint-based grammars for generation successfully.

In chapter 3 I present a generation technique, known as semantic-head-driven generation, which has some interesting properties. Firstly, the algorithm is *goal-driven* in that the order of processing is geared towards the input (semantic structures). Secondly, the algorithm is *lexicon-driven*, in that the algorithm proceeds in a bottom-up fashion. These two properties imply that the generation technique is most useful for theories, such as unification-based versions of categorial grammars (CUG, UCG, HPSG), in which semantic structures are defined in a lexical, and head-driven fashion.

I present some evidence that semantic-head-driven generation faces problems in the case of some types of discontinuous constituency. Most notably, in certain analyses of so-called 'head-movement', as in straightforward analyses of verb-second phenomena in languages such as German and Dutch, the algorithm faces severe problems. Although I discuss some restricted techniques to repair the problem, a more general solution does not seem available, for grammars which are *concatenative*.

Linguistic evidence for more powerful operations on strings than concatenation, is presented by various authors. Some proposals are discussed in chapter 4. I argue that analyses, which were problematic from a generation point of view, can be avoided in grammars which allow for more powerful operations on strings. Furthermore, as long as these operations on strings are *linear* and *non-erasing* (these terms will be explained later), it might still be possible to define efficient parsing algorithms. Chapter 4 presents such a parsing algorithm: the head-corner parser. The head-corner parser for non-concatenative grammars generalizes Martin Kay's head-driven parser, which in turn is a variation of the left-corner parser. In the head-corner parser, processing proceeds in a head-driven and bottom-up fashion. Furthermore, the bag of words occurring in the input sentence functions as the *guide*, rather than the *sequence* of words. I argue that this order of processing provides for a *goal-driven*, and *lexicon-driven* flow of control.

Summarizing, the argument can be rephrased as follows. Constraint-based grammars can be used in principle both for parsing and generation. However, to use such grammars in a practically interesting way, the grammars need to be restricted in some way. Historically, the restriction has been (in order for parsing to be efficient), that phrases are built by concatenation. No restriction for generation was assumed, as generation played a minor role. However, once grammars are to be used reversibly, I argue that this division of labor (a strict restriction for parsing, and no restriction for generation) cannot be maintained. To make generation feasible at all, some assumptions about how semantic structures are combined are necessary. In the approach of chapter 3, semantic structures are built in a lexical and head-driven fashion. In order for these assumptions to make linguistic sense, it is also necessary to have more freedom in the way phonological structures are combined. Thus, instead of concatenative grammars, linear and non-erasing grammars are called for.

In chapter 5, I describe a possible application of reversible grammars. This chapter provides motivation that such grammars can be used to implement the relation 'linguistically possible translation'. The results of the other chapters were developed partly in the context of the construction of a reversible MT prototype, called MiMo2. This prototype was developed by the author and colleagues at the University of Utrecht. In this architecture, translation is simply defined by a *series* of three reversible, constraint-based grammars.

1.1 Motivation for reversible grammars

Motivations for reversible grammars can be divided into linguistic, language technological, and psychological motivation. The different kinds of motivation are now discussed in turn.

1.1.1 Linguistic motivation

If we assume a reversible grammar, then we make two claims. The first claim is that language should be described by a single grammar (rather than a different grammar for understanding and a different grammar for production). The second claim is that this single grammar, moreover, can be used effectively both for parsing and generation.

The first claim can be motivated linguistically as follows. The primary goal of (theoretical) linguistics is to characterize languages. How such languages are used by humans (or computers) are different questions. Thus, given a language such as English, the primary goal of linguistics is to define the possible English utterances and their corresponding meanings. Thus a single language should be described by a single grammar.

The second claim, that this single grammar should moreover be (effectively) reversible, can be motivated as follows. Given that the goal of linguistics is to define the relationship between utterances and meanings, it seems that, to check a possible theory, we should be able to find out the predictions such a theory makes. That is, for a given utterance it should be possible to 'know' what the possible meanings are, according to the grammar (and vice versa). Thus, for each grammar, we want to be able to compute the corresponding meaning representations for a given utterance, and to compute the corresponding utterances for a given meaning representation.

1.1.2 Language technological motivations

In order to build practical NLP systems, the use of reversible grammars can be motivated, both on methodological ground (as a means to obtain 'better' systems) and practical grounds (as a means to obtain systems in a more efficient way).

Methodological. An important motivation for reversible grammars in NLP is of a methodological nature. If we are to use grammars both for parsing and generation we are *forced* to write grammars declaratively. This in turn implies that a more abstract analysis of the linguistic facts is necessary in the general case. If we are to write a declarative grammar which is used only for, say, parsing, it is quite easy to 'cheat' and 'adapt' the grammar to the parsing algorithm that is being used. In a reversible grammar this is much harder because at any moment there are two algorithms for which the grammar must be applicable.

The claim is that the use of reversible grammars will eventually lead to better grammars. For example, a grammar that is written for parsing will typically over-generate quite a lot; i.e. it will assign logical forms to sentences that are in fact ungrammatical. Not only is such a state of affairs undesirable if we are interested in describing the relation between form and meaning, it can also be argued that over-generation of parsing is a problem, even if we are only interested in parsing well-formed utterances, because over-generation typically leads to 'false ambiguities'.

An example may clarify this point (this example was actually encountered in the development of a working system). Consider a grammar for English that is intended to handle auxiliaries. Suppose that the English auxiliaries are analyzed as verbs that take an obligatory VP-complement. Moreover each auxiliary may restrict the *vform* (participle, infinite) of this complement. This allows the analysis of sentences such as

(1) Graham *will have been traveling* with his aunt

However, the possible *order* of English auxiliaries (eg. 'have' should precede 'be') is not accounted for and the analysis sketched above will for example allow sentences such as

(2) *Graham *will be having traveled* with his aunt

In the case of a reversible grammar such constructions should clearly not be generated, hence the analysis will be changed accordingly. However, even if the grammar is only used for parsing, this analysis runs into problems because it will assign two meanings to the sentence:

(3) Graham is having grilled meat

The meanings that are assigned, roughly correspond to the sentences:

(4) a. Graham ordered grilled meat
    b. Graham has been grilling the meat

where only the first reading is acceptable. Thus, over-generation is not acceptable, even for grammars which are used only for parsing, because over-generation typically implies that 'false ambiguities' are produced.

In some cases, the over-generation may also lead to an explosion of local possibilities during parsing. If the grammar is more constrained, then this may sometimes be good from an efficiency point of view, because in that case there are less local ambiguities the parser has to keep track of.[1]

Thus, a reversible architecture may be a useful methodology to obtain a good parsing system.

Similarly, a grammar that is built for generation will usually under-generate; i.e. it will only generate a canonical sentence for a given logical form, even if there are several possibilities. Again, from a theoretical perspective this is clearly undesirable. It can be argued that a reversible architecture also leads to a better generation system. It has often been argued that, in particular situations, a generation system should produce an un-ambiguous utterance. In other situations, however, ambiguous utterances are harmless because the hearer can easily disambiguate the utterance. In Neumann and van Noord (1992) we propose a model of language production in which a generator instructs its grammatical component whether or not it should check for ambiguity of its proposed utterance. The grammatical component, quite independently, computes an un-ambiguous utterance if so desired. For this model to be possible at all, it must

[1] Clearly this is not necessarily the case: a finite state grammar, which recognizes a superset of English need not be constrained at all, but can be parsed very efficiently ...

be the case that the grammatical component has at its disposal several utterances for a given semantic structure in order to find in a given situation the most appropriate one. Note that 'ambiguity' might be only one of several parameters that may or may not be instantiated in a given situation. Summarizing this point, the claim is that under-generation is undesirable from the point of view of extendability.

Consistency. Consider an NLP system which is used both to convey messages to a user, and to understand the requests of the user. Necessarily, the sentences the system is able to produce and to understand are somewhat limited, given the state of the art. This may not be problematic for a user, as she might adapt herself to this restriction. However, a user will invariably assume that she can use the sort of sentences the system produces itself. That is, a reasonable constraint for such a system is that the sentences it produces is a subset of the sentences it is able to understand. In a reversible grammar the problem to check that the parser and generator are *consistent* in this respect (i.e. that the system should be able to understand those types of sentences, which it produces itself) is solved automatically; hence no special consistency checking device needs to be considered.

Practical. From a practical point of view it may be argued that it is cheaper to construct one reversible grammar than two separate grammars. The same argument can then be applied to the costs of maintaining the grammars. These two arguments of course extend to the lexical entries in the grammar: in a reversible architecture only one lexicon needs to be built and maintained, although clearly non-reversible grammars may share their lexicon too.

Furthermore, a reversible grammar provides grammar writers with a very effective debugging tool. To check whether the grammar does accept ungrammatical sentences it is possible to use the generator to see whether such ungrammatical sentences are being produced. Clearly, this technique cannot be used to ensure that a grammar does not produce ungrammatical sentences, however in practice such a tool turns out to be quite useful, as many errors in the grammar are detected this way.

1.1.3 Psychological motivation

An interesting question might be whether humans base their language production and language understanding on a single body of grammatical knowledge. Clearly, this would explain why humans speak the same language they understand and vice versa. However, in practice speakers often understand sentences they would never produce. This observation may have several reasons.

Many differences are due to the fact that people often are able to understand otherwise mysterious utterances, because of the context and situation — using intelligence rather than grammatical knowledge. For example, hearers may understand an utterance even if the utterance contains a word they hear for the first time (and hence they could never have produced such an utterance), provided the situation or context makes it clear what this word means. Thus 'learning' often takes over from natural language understanding proper.

Alternatively, it may simply be the case that people understand sentences, they never utter, because they do not come up with the meaning in the first place. This situation might occur, either because they are not able to come up with the meaning (Einstein's case), or because they do not want to come up with that meaning (Dan Quayle's case) [2]. The first time that Einstein explained to his colleagues the relativity theory they were probably able to understand him. However none of them would have been *able* to produce Einstein's utterances. As another example, consider the case where someone uses special stylistic effects. A hearer may recognize the social register associated with these effects; this thus will be part of the 'meaning' of the utterances of that speaker. However, the hearer may belong to a different social class, and hence its language components will generally be instructed with a different 'meaning' representation to that effect. Thus, it seems that some of the differences between understanding and production are not to be explained linguistically, but are due to a difference at another level of cognitive behavior.

Thus, maybe it is possible to maintain that the grammatical part of language understanding and production can be modeled by assuming it is based on a reversible grammar. On the other hand, if it is not possible to maintain this claim in its full generality, then I believe that the model proposed here provides a good starting point for a more realistic model of language behavior.

1.2 Representation of utterances and meaning

I will characterize a language as a relation between 'form' and 'meaning'. In the case of spoken natural languages, the 'form' consists of 'sound'. Understanding a language means to be able to assign meaning to utterances from that language; speaking a language means to be able to produce the appropriate utterances of that language for a given meaning. In grammars, 'form' and 'meaning' need to be represented in some way or other. A grammar represents the meaning and sound between which it defines a relation, by some sort of representation. To represent natural language utterances, a grammar may define phonological representations. The denotation of these phonological representations are utterances. In current computational linguistic practice, such representations often simply take the form of a list of words, i.e. written language is used to represent spoken language. Given that written language is so common in our culture no problem arises here. If one is interested in spoken language, however, the relation between written and spoken language should be defined as well (or rather the relation between spoken language and the phonological representations).

On the other hand, there is much less agreement as to what the 'meaning' of natural language utterances is, and how that meaning should be represented. In a model theoretic view on meaning, the meaning of utterances is represented by logical formu-

[2] This characterization is due to Jim Barnett. It may be possible to understand the things D.Q. says, but one would never want to talk like him.

las. The interpretation of these formulas then constitute (model-theoretic) meaning. However, what kind of logic is needed to describe meanings of natural language is a matter of debate. I will not take part in this debate, but abstract away from the details of the choice of natural language semantics. The assumption I will make is that such logical formulas (semantic representations) can be described by feature structures. In order to provide for some exemplification, I will use simple semantic structures, which are defined in subsection 1.2.3.

1.2.1 Logical equivalence problem

It is not problematic to assume that feature structures can be used to represent semantic representations of any sort. However, it *is* problematic to assume that the constraint-language we define feature structures with, is rich enough to capture the intrinsic properties of the logic. The problem that arises in this context is called the *logical equivalence* problem (Appelt, 1987). A logic defines logical equivalences between formulas. If such equivalences are not taken into account by the grammatical formalism, unexpected results may occur. For example, consider a grammar that relates some utterance u with the meaning representation m. Suppose, furthermore, that in the logical language from which m is taken, m' is equivalent to m. What should happen if we ask a system based on that grammar to generate an utterance for the semantic structure m'? In the ideal case, the system should indeed produce u as a possible utterance. This is so, because the 'form' of the formulas should not really matter; such formulas stand for a piece of meaning, and if two formulas describe the same piece of meaning then these two formulas should behave in the same way.

However, in the general case one cannot expect from a generation system, that it is indeed capable of producing an answer in the previous example. The problem whether two expressions are logically equivalent is undecidable for any 'interesting' logic. Therefore, it is not to be expected that it is possible to devise a serious logic for natural language semantics, in which logical equivalence is decidable.

Furthermore, even simple kinds of equivalences may give rise to an enormous increase of computational complexity. For example, Calder *et al.* (1989) mention that the equivalence problem with only the axioms of commutativity and associativity for their 'Indexed Language' has factorial complexity.

Shieber (1988) mentions that, from the standpoint of natural language generation, "the class of equivalent logical forms [...] is not really closed under logical equivalence". He claims that a finer-grained notion of *intentional equivalence* is required, such that for example p and $p \wedge (q \vee \neg q)$ would not necessarily be intentionally equivalent; and these formulas might correspond to different utterances, one about p only, the other about p and q. Clearly it depends on the actual logic whether or not a different form of equivalence is called for. It may also be the case that in an appropriate natural language semantics those two formulas are not logically equivalent anyway.

Another point raised by Shieber (1988) is that even for logics in which each formula has a (unique) canonical form, a problem remains unless these canonical forms correspond exactly to those derived by the natural language grammar. Shieber mentions:

> We might use a logic in which logical equivalence classes of expressions are all trivial, that is, any two distinct expressions mean something different. In such a logic, there are no artifactual syntactic remnants in the syntax of the logical language. Furthermore, expressions of the logic must be relatable to expressions of the natural language with a reversible grammar. Alternatively, we could use a logic for which canonical forms, corresponding exactly to the natural language grammar's logical forms, do exist.
>
> The difference between the two approaches is only an apparent one, for in the latter case the equivalence classes of logical forms can be identified as logical forms of a new logical language with no artifactual distinctions. Thus, the second case reduces to the first. The central problem in either case, then, is discovery of a logical language which exactly and uniquely represents all the meaning distinctions of natural language utterances and no others. This holy grail, of course, is just the goal of knowledge representation for natural language in general; we are unlikely to be able to rely on a full solution soon.

For the reasons mentioned above, we cannot expect the generator in the previous example to produce u' in the general case. Indeed, the generation systems that have been proposed all impose a stronger condition on the generation task. Not only is the generator required to produce an utterance which is assigned an equivalent logical form by the grammar, but, moreover, the 'syntax' of the logical form must be the same as well. For a theoretical perspective on this, cf. van Deemter (1991); van Deemter (1990). This definition of the generation problem will be used throughout this thesis as well, and made precise in chapter 2.

Clearly, this is a somewhat disappointing result. Even though the problem of logical equivalence in general is undecidable, it may be the case that we can devise techniques which solve at least part of the problem; i.e. are able to recognize certain equivalences. I believe that the techniques developed in the remainder of this thesis can be augmented with such techniques once these become available, as follows. In a constraint-based grammar logical forms are defined by constraints. In the constraint-language to be defined in chapter 2, the only equivalence relation is the one we obtain by the equality of our constraint language. However, the general way in which I define a constraint-based formalism, along the lines of Höhfeld and Smolka (1988), allows for more powerful constraints, if appropriate constraint-solving mechanisms are available.

The principal way to enrich the equivalence notion in a constraint-based formalism is to enrich the constraint-language. Thus, if we take $p \oplus q$ to be equivalent to $q \oplus p$, then we should add some axioms to the constraint language to this effect. The result of this approach will then be that $p \oplus q$ will 'unify' with $q \oplus p$. Note that the parsing- and generation algorithms defined in this thesis can be viewed as abstracting away from the particular constraint-language that is being used. Thus, given appropriate constraint-solving techniques (unification algorithms), more powerful constraints can easily be imported.

The point then is, that even though we have not much to say about the logical equivalence problem, we argue that the results of this thesis are not dependent on the way in which this problem is solved eventually. This is so, because on the one hand we abstract away from the actual choice of semantic representations, and on the other hand provide for a hook in which equivalences can be defined (in the constraint-language); the techniques developed in this thesis can easily be adapted to more powerful constraint-languages as well.

1.2.2 Unification-based semantics

In constraint-based grammars, of the sort I will be assuming throughout, semantic structures are usually composed by constraint-solving, rather than by functional application (with lambda expressions and lambda reductions). For the simple constraint language, which consists solely of path-equations, which I define in the next chapter, this comes down to a unification-based semantics. This technique has become quite popular in work on computational semantics. A theoretical perspective on the use of unification to construct semantic structures, is presented in Moore (1989).

Motivation for the use of unification, rather than functional application, using complex lambda expressions and lambda reductions, partly is of a computational kind. For example, van Eijck and Moore (1992), state that:

> It is simpler and more efficient to use the feature system and unification to do explicitly what lambda expressions and lambda reduction do implicitly, that is, assign a value to a variable embedded in a logical form expression.

and Alshawi and Pulman (1992) furthermore notice, when comparing unification-based semantics with more traditional semantics, for the purpose of generation:

> [...] If the semantic rules had been more in the style of traditional Montague semantics, generation from structures that had undergone lambda reductions would have presented search difficulties because the reductions would have to be applied in reverse during the generation process. This turns out to be an important practical advantage of unification-based semantics over the traditional approach.

Nerbonne (1992) also discusses some advantages of constraint-based semantics. Using constraints to define semantic structures is motivated because it allows that relations with syntax, phonology, and context can be stated in a simple way, by the interaction of the constraints from the different domains. Nerbonne (1991) claims:

> There are several advantages of the unification-based view of the syntax/semantics interface over the more familiar (Montagovian) view of this interface, which is characterized by a homomorphism from syntax into semantics. The unification-based view sees the interface as characterized by a set of constraints to which non-syntactic information may contribute, including phonological and pragmatic information. Let the semantics of intonation and that of deixis serve as examples of the two sorts. The

feature-based view furthermore allows syntactic and semantic information to be bundled in complex, but useful ways.

Other examples and discussion of the use of constraints in order to define the semantics, are for example Fenstad *et al.* (1987), Pereira and Shieber (1987), Halvorsen and Kaplan (1988), Rupp (1991).

1.2.3 Example semantic structures

The techniques to compute the relation between phonological and semantic structures (in parsing and generation), and between semantic structures of different languages (in machine translation) are supposed to abstract away from the particularities of these phonological and semantic structures. However, for concreteness and expository purposes I will define very simple semantic structures which are used throughout the example grammars and rules in this thesis. These semantic structures are feature structures, of the sort to be introduced in chapter 2. The reader unfamiliar with feature structures is advised to consult that chapter first.

The semantic structures to be described here are essentially predicate argument structures decorated with some syntactic features. Similar structures were used in the MiMo2 translation prototype (this prototype is described in van Noord *et al.* (1990); van Noord *et al.* (1991), and in chapter 5). Semantic structures come in several 'sorts'. A semantic structure has a label *sort* of which the value is one of

$$\{\text{nullary}, \text{unary}, \text{binary}, \text{ternary}, \text{modifier}\}$$

Other attributes of semantic structures include *pred*, *mod*, *arg1*, *arg2* and *arg3* of which the values are semantic structures themselves. The convention is that structures with sort 'ternary' are specified for *arg1*, *arg2* and *arg3*, whereas structures with sort 'binary' are not specified for *arg3*, and so on. The attribute *pred* takes constants as its values (these constants often represent content words). Furthermore semantic structures are decorated with labels such as *neg*, *number*, *tense*, *aspect*, *def*,... of which the values are atomic and of which the intention will be clear. These syntactic labels play a minor role in this thesis, and are often left out. For example, the sentence 'the priest drinks wine' is associated with argument structure:

$$\begin{bmatrix} \textit{sort} : \text{binary} \\ \textit{pred} : \text{drink} \\ \textit{arg1} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{priest} \\ \textit{number} : \text{sg} \\ \textit{def} : \text{def} \end{bmatrix} \\ \textit{arg2} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{wine} \\ \textit{def} : \text{indef} \\ \textit{number} : \text{mass} \end{bmatrix} \\ \textit{tense} : \text{present} \\ \textit{neg} : \text{nonneg} \end{bmatrix}$$

Furthermore, modifier structures such as noun-adjective constructions are represented by a semantic structure of sort 'modifier', with labels *mod* and *arg1* of which the values are semantic structures. Therefore, the semantic structure of a noun phrase 'very strong whisky', may look as follows:

$$\left[\begin{array}{l} \mathit{sort}: \text{modifier} \\ \mathit{mod}: \left[\begin{array}{l} \mathit{sort}: \text{modifier} \\ \mathit{mod}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{very} \end{array}\right] \\ \mathit{arg1}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{strong} \end{array}\right] \end{array}\right] \\ \mathit{arg1}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{whisky} \\ \mathit{def}: \text{indef} \\ \mathit{number}: \text{mass} \end{array}\right] \end{array}\right]$$

As a more complex example the logical form of

(5) The soldiers did not open fire on the Columbian prime minister

is the following feature structure:

$$\left[\begin{array}{l} \mathit{sort}: \text{binary} \\ \mathit{pred}: \text{open\_fire\_on} \\ \mathit{arg1}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{soldier} \\ \mathit{def}: \text{def} \\ \mathit{number}: \text{pl} \end{array}\right] \\ \mathit{arg2}: \left[\begin{array}{l} \mathit{sort}: \text{modifier} \\ \mathit{mod}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{columbian} \end{array}\right] \\ \mathit{arg1}: \left[\begin{array}{l} \mathit{sort}: \text{modifier} \\ \mathit{mod}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{prime} \end{array}\right] \\ \mathit{arg1}: \left[\begin{array}{l} \mathit{sort}: \text{nullary} \\ \mathit{pred}: \text{minister} \\ \mathit{def}: \text{def} \\ \mathit{number}: \text{sg} \end{array}\right] \end{array}\right] \end{array}\right] \\ \mathit{neg}: \text{neg} \end{array}\right]$$

To encode for example control relations I introduce a special semantic structure of which the sort is 'refer' and of which the only other attribute is *index*. Furthermore, other semantic structures may also be specified for the *index* attribute. Sharing of the index attribute then can be used to indicate control, as in the following example for the sentence

(6) The soldiers try to shoot the whisky priest

$$\begin{bmatrix} \textit{sort}: \text{binary} \\ \textit{pred}: \text{try} \\ \textit{arg1}: \begin{bmatrix} \textit{sort}: \text{nullary} \\ \textit{pred}: \text{soldier} \\ \textit{number}: \text{pl} \\ \textit{index}: \text{I} \end{bmatrix} \\ arg2: \begin{bmatrix} \textit{sort}: \text{binary} \\ \textit{pred}: \text{shoot} \\ \textit{arg1}: \begin{bmatrix} \textit{sort}: \text{refer} \\ \textit{index}: \text{I} \end{bmatrix} \\ \textit{arg2}: \begin{bmatrix} \textit{sort}: \text{nullary} \\ \textit{pred}: \text{whisky\_priest} \\ \textit{number}: \text{sg} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In examples throughout this thesis I will often abbreviate the semantic structures presented above (somewhat informally for expository purposes) as follows. Semantic structures of which the *sort* attribute is 'nullary' are abbreviated by Pred where Pred is the (atomic) value of the *pred* attribute. Semantic structures of type 'unary', 'binary', and 'ternary' are abbreviated resp. by Pred(Arg1), Pred(Arg1,Arg2) and Pred(Arg1,Arg2,Arg3) where Pred is the (atomic) value of the *pred* attribute and Arg1, Arg2 and Arg3 are resp. the (abbreviated) values of the *arg1*, *arg2* and *arg3* attributes. Semantic structures of sort 'modifier' are abbreviated as [Mod](Arg1) where Mod is the abbreviation of the value of the *mod* attribute, and Arg1 the abbreviation of the value of the *arg1* attribute. Semantic structures of sort 'refer' are abbreviated by *ref*. Finally, for each of these abbreviations, if the value I of their corresponding *index* attribute occurs more than once in a structure, the abbreviated semantic structure is prefixed with I:. The values of other attributes will be abstracted away from in such abbreviated semantic structures.

As an example, I write the semantic structure corresponding to 'The soldiers tried to shoot the very brave columbian minister' as

try(I:soldier,shoot(I:ref,[[very](brave)]([columbian](minister))))

1.3 Reversibility

Intuitively, we call a program 'reversible' if it is capable of both parsing and generation on the basis of a single characterization of the relation between semantic structures and phonological structures. The following definitions of *reversibility* are meant to be independent of the way we go about achieving a reversible natural language processing component. Furthermore, the definitions abstract away from the actual representations between which we are defining relations. In chapter 5 we propose to use constraint-based grammars for a transfer component of an MT system. In that case the relation defined by the grammar is between semantic representations of different languages; the following definitions are generalized in order to be applicable for such usages as well.

A program or system will be called *r-reversible* iff it computes a binary relation r in both directions. The idea is that, given an element of a pair in the relation, the program computes the corresponding element(s) of that pair. To encode the 'direction' of the relation I assume that the input for the program consists of a pair $\langle dir, x \rangle$ where dir represents the direction which the program should compute. The value of dir is either 0 or 1. If the value is 0, then the program computes the relation from left to right; if the value is 1 then the program computes the relation from right to left. [3]

Definition 1 (Compute a relation in both directions) A program P computes a relation r in both directions, iff P enumerates for a given input $\langle dir, e \rangle$ the set

$$\{x| \begin{array}{l} \langle e, x \rangle \in r \wedge dir = 0 \vee \\ \langle x, e \rangle \in r \wedge dir = 1 \end{array} \}$$

Definition 2 (Reversible)

- A program P is *r-reversible* iff P computes r in both directions.
- A relation r is *reversible* iff there exists an r-reversible program.

Consider the case where r is the relation between phonological and semantic representations defined by some grammar. In this case a program is said to compute this relation in both directions (i.e. the program is reversible w.r.t. this relation) iff for a given phonological representation the program returns the corresponding semantic representation; for a given semantic representation the program returns the corresponding phonological representations. Such a program may consist of a parser and a generator (depending on the value of dir), or alternatively the program consists of a single uniform algorithm. For example in the case of the meta-interpreter for $\mathcal{R}(\mathcal{L})$-grammars to be presented in chapter 2, the value of dir defines to which path (eg. *phon* or *sem*) the input has to be assigned, otherwise the parser and generator are equivalent.

Systems in which the relation between phonological and semantic representations is defined procedurally are seldom reversible in this respect, because it is very difficult to make sure that the program indeed computes the same relation in both directions. On the other hand, a system based on a single *declarative grammar* necessarily is reversible.

Arguably, the above defined notion of reversibility is somewhat weak. According to the definition above, any recursively enumerable relation is reversible. However, in practice it is often the case that a grammar that is developed from a single perspective (eg. parsing perspective) is completely useless in the other direction because it

[3]Note that it usually will be quite clear from the input in which direction the relation is to be computed.

simply fails to terminate in all interesting cases (let alone efficiency considerations). Therefore, I define what it means for a relation to be *effectively reversible*. A relation is effectively reversible if it can be effectively computed in both directions, i.e. there exists a program computing the relation in both directions, and furthermore the program always halts. In the terminology of Hopcroft and Ullman (1979), we require that there exists an *algorithm* computing the relation.

Definition 3 (Effectively reversible)

- A program P is effectively r-reversible iff
  - P is r-reversible; and
  - P is guaranteed to terminate (for every input).
- A relation r is *effectively reversible* iff there exists an effectively r-reversible program.

If I use the term *reversible* in the remainder of this thesis, then I will invariably mean *effectively reversible*.

Next I show that the composition of (effectively) reversible relations is (effectively) reversible. This proposition motivates the use of a series of grammars, each defining an (effectively) reversible relation, to obtain an (effectively) reversible MT system (chapter 5).

Definition 4 (Composition) The composition of two relations $r_1 \circ r_2$ is the relation $\{\langle a, c\rangle | \langle a, b\rangle \in r_1 \text{ and } \langle b, c\rangle \in r_2\}$.

Proposition. The composition of two effectively reversible relations is effectively reversible.

Proof. Assume r and r' are reversible. We need to show that $r \circ r'$ is reversible. Let P and P' compute resp. r and r' in both directions. Construct the program as in figure 1.1 which computes the series of P and P', the order of which is dependent on the direction. Each element of the output of the first program is taken as the input to the second program. As both P and P' terminate, the two programs in series terminate too. Another, intuitively more attractive way to think about the above construction is pictured in figure 1.2.

1.4 Overview

1.4.1 Towards reversible grammars

An interesting goal for this thesis might have been to devise an interpreter for reversible grammars. This interpreter should indeed compute the relation r for any given r-reversible grammar. However, it is not clear that such an interpreter actually can be

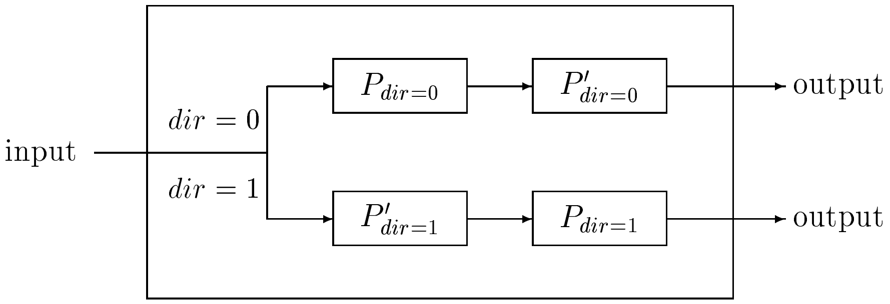


Figure 1.1: Program computing composition.

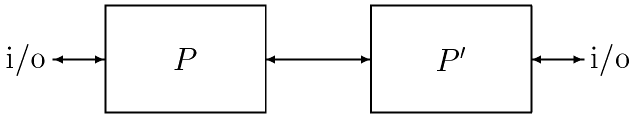


Figure 1.2: Series of reversible programs

built. Even though for each of the r-reversible grammars, programs can be built that compute r, it is not clear that a universal program can be built that accomplishes that task for any given grammar. In other words, even though the p-parsing problem (this notion will be defined in chapter 2) is solvable, by definition, for a fixed reversible grammar, it is not clear that the universal p-parsing problem for reversible grammars is solvable.

The goal of this thesis is more modest. Most attention in the next chapters will be devoted to the development of parsing and generation techniques which improve upon existing techniques with respect to the following two, related, dimensions:

- Applicability. The parsing and generation techniques are applicable for a larger class of constraint-based grammars than some other techniques. Furthermore, this extended domain of applicability is motivated from a linguistic perspective. Thus, the proposed techniques are applicable for constraint-based grammars such as these are actually written by (computational) linguists.

- Linguistic Deduction. The parsing and generation techniques follow certain linguistic principles. That is, the techniques are motivated from a linguistic perspective. It is hoped that such linguistically motivated techniques improve upon the efficiency as compared with other deduction methods.

Chapter 3 discusses methods for (grammatical) generation on the basis of constraint-based grammars. A method for generation, called 'semantic-head-driven' generation, is introduced and compared with some 'top-down' generation techniques, as for example proposed by Wedekind (1988) and Dymetman and Isabelle (1988), and with the chart-based technique of Shieber (1988). With respect to the dimensions mentioned above, semantic-head-driven generation is motivated because:

- Applicability. Certain linguistically motivated left-recursive analyses are problematic for the approaches of Wedekind (1988) en Dymetman and Isabelle (1988), but are handled without problems in a semantic-head-driven generator. Furthermore, certain analyses of idiomatic constructions are impossible in Shieber's chart-based generator. These analyses pose no problem for semantic-head-driven generation.

- Linguistic Deduction. The semantic-head-driven generation strategy defines a mixed bottom-up and top-down search procedure. The search is guided by the input semantic structure through the use of the notion 'semantic-head of a construction'. Furthermore, the search is guided as much as possible by the information available in lexical entries. This implies that the algorithm is most useful for constraint-based grammars based on lexicalistic linguistic theories with a constraint-based semantics, such as UCG and HPSG.

Chapter 4 is devoted to parsing. A method for parsing, called 'head-corner' parsing is introduced. Again, this method is compared with competing approaches to parsing along the dimensions of applicability and linguistic deduction:

- Head-corner parsing is applicable for constraint-based grammars in which operations on strings are restricted to be non-erasing and non-copying. One such operation is concatenation, but there are many others. For that reason head-corner parsing is applicable for a strict superset of concatenative constraint-based grammars, whereas most other algorithms are applicable only for concatenative grammars. As an example it is shown how head-corner parsing can be employed to parse lexicalized and constraint-based versions of Tree Adjoining Grammars.

- The order of processing in the head-corner parsing is bidirectionally in two senses. Firstly, the parsing proceeds head-driven, rather than from left-to-right. This implies that powerful top-down predictions are possible based on the usual percolation of syntactic features between the mother and the (syntactic) head of a construction. In many linguistic theories much syntactic information is encoded in the lexicon. As the head-corner parser has an important flow of bottom-up information, this lexical information is used to further reduce the size of the search space. The combination of these two properties furthermore ensures that, once the head of a construction is known, the parser also knows what other phrases it should expect, using the subcategorization specifications of that head.

1.4.2 The other chapters

The next chapter provides the basis of the other chapters of this thesis, by defining a constraint-based formalism, called $\mathcal{R}(\mathcal{L})$. The constraint-language of the formalism consists of the path-equations known from PATR II (Shieber *et al.*, 1983). This may seem somewhat restricted, as lately some results have been obtained in the area of disjunctive and negative constraints (see for example Johnson (1988) and Smolka (1989);

for an overview of other types of constraints cf. Wedekind *et al.* (1990)). However, the way in which we define $\mathcal{R}(\mathcal{L})$, along the lines of Höhfeld and Smolka (1988), should make it clear that more powerful constraint languages can be exchanged with the simple constraint language we are assuming, provided appropriate constraint-solving techniques are available for such more powerful constraint languages. Therefore, the results presented in the other chapters of this thesis can easily be generalized to more powerful constraint languages.

On the other hand, $\mathcal{R}(\mathcal{L})$ is more general than most grammar formalisms, in that it does not prescribe that phrases are combined by concatenation. No assumptions about string combination are enforced in the formalism. The motivation for this enrichment is provided by certain analyses of *discontinuous constituency* constructions, which employ other possible combinations of strings. Further motivation is provided in chapter 3, where it turns out that certain analyses in concatenative grammars are problematic for generation. These problems can be solved in non-concatenative grammars. In chapter 4 the extra power of non-concatenative grammars will be investigated and exploited. Furthermore, this more general perspective allows the use of the formalism for other tasks as well. In chapter 5 the formalism is used to define transfer grammars in a machine translation system.

The formalism defined in chapter 2 is used in this thesis in two ways. Firstly, and most importantly, we use the formalism to define *grammars* with. But furthermore, we use the formalism to define *meta-interpreters* for such grammars in. For this reason the formalism is provided with a simple procedural semantics (essentially as in Prolog). However, a main theme of this thesis will be that for 'linguistic deduction' (parsing and generation) different proof strategies are more appropriate. These will then be defined in $\mathcal{R}(\mathcal{L})$ as meta-interpreters.

Constraint-based grammars can be used, in principle, both for parsing and generation. This is also true, for grammars defined in $\mathcal{R}(\mathcal{L})$. However, to use such grammars in a practically interesting way, the grammars need to be restricted in some way. Historically, the restriction has been (in order for parsing to be efficient), that phrases are built by concatenation. No restriction for generation was assumed, as generation played a minor role. But, generation on the basis of declarative grammars is not without problems. For example, the simple top-down, left-to-right backtrack search strategy leads to non-termination for linguistically motivated grammars. Chapter 3 discusses several (linguistically relevant) problems for some obvious generation techniques. Furthermore, that chapter provides motivation for a different processing strategy, in which generation proceeds essentially *bottom-up*, and *head-driven*. A simple version of this strategy is defined, and its properties and short-comings are investigated. Several variants and possible improvements are discussed. Relying on the notion 'head', has some implications for the way in which semantic structures should be combined in the grammar. Essentially, the semantic-head-driven generation algorithm embodies the assumption that semantic structures are defined in a *lexical* and *head-driven* fashion. It turns out that the head-driven generation strategy faces problems with analyses in which this assumption is violated. As an important example, the semantic-head-driven generation strategy faces problems, if the head of a construction has been 'displaced'. Such an analysis is often assumed for verb-second phenomena in for example German and Dutch, if the grammars are restricted to be concatenative. Thus, the assumption that semantic structures are built lexically and head-driven, does not make linguistic sense, if phrases are to be built by concatenation. In order for these assumptions to make linguistic sense, it is therefore necessary to have more freedom in the way phonological structures are combined. Thus, instead of concatenative grammars, non-concatenative grammars are called for.

In chapter 4 linguistic motivation for such powerful operations is discussed. A number of proposals for operations like 'head-wrapping' and 'sequence-union' are described, and a class of formalisms is introduced in which operations on strings are allowed which are *linear* and *non-erasing*. Formalisms such as Head-Grammars (Pollard, 1984) and Tree Adjoining Grammars (Joshi *et al.*, 1975), are members of this class. Clearly, parsing algorithms developed for concatenative formalisms cannot be used for these more powerful formalisms. An important question thus is how to parse with non-concatenative grammars. I describe a very general algorithm for this class of grammars which proceeds, again, *bottom-up*, and *head-driven*. The motivation for such a processing strategy is discussed. Furthermore, I describe possible extensions and improvements of the basic algorithm. I also show how the parser can be 'specialized' for lexicalized, and constraint-based, versions of Tree Adjoining Grammars.

In chapter 5, I describe a possible application of reversible grammars. This chapter provides evidence that such grammars can be used to implement a machine translation system. The results of the other chapters in fact were developed partly in the context of the construction of a reversible MT system, called MiMo2. In this chapter I discuss the notion 'linguistically possible translation' On the basis of this discussion a specific architecture for an MT system is proposed, which has been implemented as the MiMo2 prototype. This prototype was developed by the author and colleagues at the University of Utrecht. In this architecture, translation is simply defined by a *series* of three reversible, constraint-based grammars. It is shown how $\mathcal{R}(\mathcal{L})$-grammars can be used to define 'transfer' relations as part of a Machine Translation system. Furthermore, a constraint on transfer grammars is proposed that ensures termination, while certain context-sensitive translations are still possible. It is argued that reversible transfer grammars provide an interesting compromise between expressive power and computational feasibility.

Most of the material in this thesis is based on papers and articles that have appeared elsewhere. The material on translation is partly based on van Noord *et al.* (1990); van Noord (1990b); van Noord *et al.* (1991). The chapter on generation is based on van Noord (1989); Shieber *et al.* (1989); van Noord (1990a); Shieber *et al.* (1990), and the material in the chapter on parsing has been published as van Noord (1991b); van Noord (1991a).

Chapter 2

A Powerful Grammar Formalism

The grammar formalism I will define in this chapter is closely related to other formalisms currently in use in computational linguistics. These formalisms are known as 'unification-based', 'constraint-based', 'information-based' and 'feature-logic based'. Members of this class are for example Definite Clause Grammars (Pereira and Warren, 1980), PATR II (Shieber *et al.*, 1983), Functional Unification Grammar (Kay, 1985) and formalisms underlying linguistic theories such as Generalized Phrase Structure Grammar (Gazdar *et al.*, 1985), Lexical Functional Grammar (Bresnan, 1982), Unification Categorial Grammar (Zeevat *et al.*, 1987), Categorial Unification Grammar (Uszkoreit, 1986) and Head-driven Phrase Structure Grammar (Pollard and Sag, 1987).

A general characterization of such constraint-based formalisms is given in Höhfeld and Smolka (1988). They also show how the nice properties of logic programming languages carry over to a whole range of such constraint-based formalisms, by abstracting away from the actual constraint language that is used. I define such a constraint-based formalism in which the underlying constraint language, $\mathcal{L}$, consists of path equations. The most important characteristics of the formalism are:

- The formalism consists of definite clauses, as in Prolog; instead of first-order *terms* the data structures of the formalism are *feature structures*.
- The formalism does not assume that concatenation is the sole string-combining operation (in contrast to FUG, DCG, PATR II, LFG, GPSG and UCG).
- The formalism is defined in an abstract framework, which facilitates the extendability of the techniques I develop in later chapters, to formalisms based on other (more powerful) constraint-languages.

Each of these points will now be clarified in turn.

Firstly, the principal 'data-structures' of the formalism are feature structures, rather than first-order terms such as in Prolog. The motivation is that such feature structures are closer to the objects usually manipulated by linguists. Furthermore, in writing grammars the use of first-order terms becomes rather tiresome because it is necessary to keep track of the number of arguments functors take, and the position of sub-terms in such terms. Using path equations to define feature structures achieves some sort of data abstraction. As an example consider a program which manipulates terms such as the following

```
sign(syn(Loc,Sc,head(Agr,Cat)),sem(Pred,Args),phon(In,Out))
```

and suppose furthermore we want to refer to a specific part Cat of such a term T. In Prolog we are then forced to mention all intermediate functors, and for each functor we need to mention all its arguments (possibly using the 'anonymous' variable '_'):

```
sign(syn(_,_,head(_,Cat)),_,_)
```

On the other hand, using path equations, it is possible to refer to such an embedded term by its *path*, in this case the value of Cat is obtained by the equation Cat $\doteq$ T *syn head cat*.

This said, it should be stressed though that the difference between the two approaches is not very decisive. In fact first-order terms may be used in an implementation of such graph-based formalisms (Hirsch, 1988), and data-abstraction can also be achieved by other means such as syntactic macro's, or by auxiliary predicates.

From a linguistic point of view, the second characteristic is the most salient one. The formalism to be proposed, does not enforce that concatenation is the sole operation to combine strings. This choice can be motivated by:

- Increased symmetry of parsing and generation
- Increased expressive power
- Other applications

Dropping the concatenative base can be motivated from the desire to use grammars in a *reversible* way. From a reversible viewpoint, it is attractive to view a grammar simply as a definition of the relation between strings and logical forms. To give a different status to the phonology attribute, seems to destroy the inherent symmetry somewhat. Thus, the formalism does not prescribe how the value of the phonology attribute is to be composed, just as it does not prescribe how the value of the semantics attribute is composed.

If we do not incorporate a concatenative base, then we allow for investigation of other types of string combinations in natural language grammars. Several researchers, see for example Bach (1979), Pollard (1984), Reape (1989) and Dowty (1990), have noted that analyses of a whole range of linguistic phenomena (most notably those involving discontinuous constituents) may be simplified by assuming other types of string operations. Some of the proposals are discussed in chapter 4.

As I will demonstrate below, if no assumptions about the construction of phonological representations, or semantic representations, are defined, then the parsing and generation problem of the formalism is generally not decidable. An important theme of this thesis is, to investigate parsing and generation procedures which can be applied usefully, for linguistically motivated grammars. In chapter 4 I describe a parsing algorithm for a subset of $\mathcal{R}(\mathcal{L})$, in which strings are constructed by operations which are to *linear* and *non-erasing*. This algorithm thus imposes some requirements on the ways strings are built, but these requirements are less restrictive than concatenation.

Another reason for developing a formalism which is not based on concatenation is the observation that other (non-linguistic) problems can be encoded in a unification-grammar as well, if we are not forced to manipulate strings. In chapter 5 I show how the resulting formalism can be used to define relationships between language specific logical form encodings (chapter 5), as the transfer component of a machine translation system. Furthermore, the formalism is also used to define meta-interpreters in — this usage of the formalism also entails that no assumptions about string construction are built-in.

The third characteristic states that the resulting formalism is a member of a class of constraint-based formalisms. Therefore, results that hold for this class carry over to the present formalism. In the other direction, it is easy to see how the current formalism can be *extended* to allow for other, perhaps more complex constraints. This is very useful as in the last few years a whole family of different constraints have been proposed that do extend formalisms such as PATR II. An overview of these proposals can be found in Wedekind *et al.* (1990). The formalization of Höhfeld and Smolka (1988) makes it possible to see how these extended formalisms belong together. In a somewhat idealized view, parsing and generation algorithms defined for a member of the class of constraint-based formalisms, can be used for other members of this class, provided the appropriate *constraint-solving* techniques are available for the constraints incorporated in these other formalisms. For example, the generation- and parsing algorithms to be presented in the chapters 3 and 4, can easily be used for formalisms that include versions of negation and disjunction.

2.1 The constraint language: $\mathcal{L}$

In this section I define $\mathcal{L}$ as the underlying constraint language of $\mathcal{R}(\mathcal{L})$. The definition is based on Shieber (1989) and Smolka (1989). This constraint language is used to describe structured objects called 'feature structures'. Its semantics will be defined with respect to sets of 'feature graphs', which are directed, connected, labelled graphs.

2.1.1 Constraints

The formulas of the logic (*constraints*) are built from a set V of variables, a set C of constants and a set L of labels (also called *attributes* or *features*). Variables are thought of as referring to some specific feature structure. The labels are thought of as pointing to a specific part of a feature structure. A *path* will be a (possibly empty) sequence of such labels. Such a path can be viewed as the 'address' of some piece of information in a specific feature structure. For example the path *syn agr person* will point to the *person* part of the *agr* part of the *syn* part.

A descriptor is a sequence sp where s is either a variable or a constant, and p is a (possibly empty) path. I write ϵ to refer to empty paths. Examples of descriptors are

$$\mathrm{X}_0 l_1 l_2$$
$$\mathrm{c} l_1$$
$$\mathrm{X}_0$$

assuming that X_0 is a variable, l_i are labels and c is a constant. [1]

Atomic constraints (also called *path equations*) are equations of the form:

$$d_1 \doteq d_2$$

where d_1 and d_2 are descriptors. For example, the following expressions are atomic constraints, assuming that $\{\mathrm{X}_1, \mathrm{X}_2\} \subseteq V$, $\{syn, agr, number\} \subseteq L$ and $\{\text{singular}\} \subseteq C$:

$$\mathrm{X}_1 \ syn \ agr \ number \doteq \text{singular}$$

$$\mathrm{X}_1 \ syn \ agr \doteq \mathrm{X}_2 \ syn \ agr$$

An *$\mathcal{L}$-constraint* is a set of such atomic constraints, i.e. a set of path equations. The equations in such a constraint are interpreted conjunctively. I write such a constraint as a sequence of equations separated by commas:

$$\phi_1, \ldots, \phi_n$$

In the following, d is used to refer to descriptors in general, X_i refers to variables, l_i refers to labels, c_i refers to constants, s and t refer to either constants or variables and p and q refer to paths.

2.1.2 Feature graphs

The semantics of $\mathcal{L}$ employing labels L, constants C, and variables V is defined with respect to the domain of *feature graphs*, built from L, C, and V. A feature graph is a directed graph which is finite, rooted, and connected. Furthermore, the labels of the edges leaving a node must be pairwise distinct. Every inner node of a feature graph is a variable, and every terminal node is either a variable or a constant. Edges are triples $\mathrm{X}lt$ such that X is a variable, l is a label and t is either a variable or a constant.

Definition 5 (Feature graphs) A feature graph is

- a pair $(\mathrm{c}, \emptyset)$ where c is a constant and $\emptyset$ is the empty set; or

[1] Note that, unlike in PATR II, the descriptor $\mathrm{c}l$ is syntactically allowed, but all constraints containing such descriptors will be unsatisfiable. Such descriptors are allowed because this allows a more elegant definition of the simplification rules defined later.

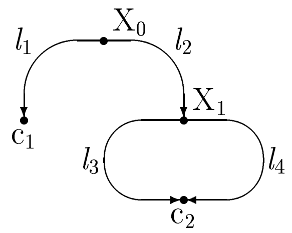


Figure 2.1: Feature graph F_1

- a pair (X_0, E) where X_0 is a variable (the root) and E is a set of edges such that
  - if Xls and Xlt are both in E then $s = t$; i.e. labels are functional; and
  - if Xls is in E then X is reachable from X_0 by the edges in E; i.e. feature graphs are connected.

Note that this definition implies that no edges leave constant nodes, because edges always start with a variable. A node s is reachable from a node t in a feature graph F iff $t \rightarrow_F^* s$ where $\rightarrow_F^*$ is the transitive and reflexive closure of $\rightarrow_F$ which is a binary relation on the nodes occurring in F:

$t \rightarrow_F s$ iff $tls \in$ the set of edges of F

The following notation to traverse feature graphs is introduced. If F is a feature graph and p is a path, then F/p is the variable or constant that is reached from the root of F where the labels in the path correspond to the labels of the edges. For example consider the following feature graph

$$F_1 = (X_0, \{X_0 l_1 c_1, X_0 l_2 X_1, X_1 l_3 c_2, X_1 l_4 c_2\})$$

which is graphically represented in figure 2.1. The expression $F_1/l_2 l_4$ denotes c_2; and $F_1/l_2 = X_1$. Furthermore, F_1/l_3 is undefined.

More formally, for $p = l_1 \ldots l_k$ a path, and t a node (variable or atom), define t/p to be the node given as follows. If $k = 0$ then $t/p =_{def} t$. Otherwise, $t/l_1 \ldots l_k$ is defined to be the node t' if there is an edge labelled l_1 from t to t'' and $t' = t''/l_2 \ldots l_k$ (otherwise $t/l_1 \ldots l_k$ is undefined). Furthermore, for F a feature graph with root X_0, $F/p =_{def} X_0/p$.

A feature graph F is called a *subgraph* of a feature graph F' if the root of F is a variable or atom occurring in F' and every edge of F is an edge of F'. If F is a feature graph and s a node of F, then F^s denotes the unique maximal subgraph of F of which the root is s. In that case, F^s has as its root the node s, and as its edges are all those edges of the form tlt' present in F such that both t, t' are reachable from s in F. For example, if F_1 is the feature graph in figure 2.1, then we have:

$$F_1^{X_1} = (X_1, \{X_1 l_3 c_2, X_1 l_4 c_2\})$$

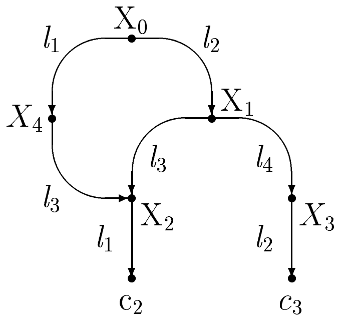


Figure 2.2: The feature graph F_2

Furthermore, for F a feature graph and p a path, F^p denotes the subgraph F^s where $F/p = s$, if this is defined. For example:

$$F_1^{l_2} = (\mathrm{X}_1, \{\mathrm{X}_1 l_3 \mathrm{c}_2, \mathrm{X}_1 l_4 \mathrm{c}_2\})$$

Otherwise F^p is undefined.

2.1.3 Solutions of constraints

An interpretation $\mathcal{I}$ of $\mathcal{L}$ consists of a domain $D^{\mathcal{I}}$ which is the set of all feature graphs built from L, C and V, and a solution mapping $\cdot^{\mathcal{I}}$ which will be defined below. An $\mathcal{I}$-assignment is a mapping from the set of variables to $D^{\mathcal{I}}$. I write $ASS^{\mathcal{I}}$ for the set of all $\mathcal{I}$-assignments. Variables and constants will denote feature graphs, relative to some assignment.

The denotation of a variable X w.r.t. assignment α is simply $\alpha(\mathrm{X})$. The denotation of a constant c is the feature graph $(\mathrm{c}, \emptyset)$ (for any assignment). The denotation of a descriptor sp is defined as F^p where F is the denotation of s; i.e. the denotation of sp is the subgraph at p of the graph denoted by s. Note that the denotation of some descriptors is undefined. Summarizing:

$$\mathrm{c}_\alpha^{\mathcal{I}} =_{def} (\mathrm{c}, \emptyset)$$
$$\mathrm{X}_\alpha^{\mathcal{I}} =_{def} \alpha(\mathrm{X})$$
$$(sp)_\alpha^{\mathcal{I}} =_{def} (s_\alpha^{\mathcal{I}})^p$$

As an example, consider the feature graph F_2 in figure 2.2. For an assignment that maps X to the feature graph F_2, the denotation of the descriptor $X l_1 l_3$ is the subgraph rooted at X_2. Similarly, the denotation of $X l_2 l_4 l_2$ is the feature graph $(c_3, \emptyset)$. Note that the denotation of the descriptor c (i.e. $\mathrm{c}\epsilon$) is the feature graph $(\mathrm{c}, \emptyset)$.

An interpretation $\mathcal{I}$ *satisfies* an atomic constraint $\phi = d_1 \doteq d_2$, relative to an assignment α, written $\mathcal{I} \models_\alpha \phi$ if the denotation of descriptors d_1, d_2 are both defined and the same, i.e.:

$$\mathcal{I} \models_\alpha d_1 \doteq d_2 \text{ iff } (d_1)_\alpha^{\mathcal{I}} = (d_2)_\alpha^{\mathcal{I}}$$

Hence, $\mathcal{I}$ satisfies the equation $X l_1 l_3 \doteq X l_2 l_3$ with respect to the assignment that maps X to the feature graph F_2 defined above. As another example, $\mathcal{I}$ also satisfies the equation $X l_1 l_3 l_1 \doteq \mathrm{c}_2$, with the same α.

The *solutions* of a constraint are all assignments that give satisfaction. Constraints are thus seen as restrictions on the values the variables in the constraints can take. The solutions of an atomic constraint ϕ are defined as follows:

$$\phi^{\mathcal{I}} =_{def} \{\alpha \in ASS^{\mathcal{I}} | \mathcal{I} \models_{\alpha} \phi\}$$

The set of solutions of a constraint $\phi_1, \ldots, \phi_n$ is defined as the intersection of the solutions of its atomic constraints:

$$(\phi_1, \ldots \phi_n)^{\mathcal{I}} =_{def} \bigcap_{1 \leq i \leq n} \phi_i^{\mathcal{I}}$$

A constraint is *satisfiable* iff it has at least one solution. Two constraints are *equivalent* iff they have the same solutions. A constraint is *valid* iff its solutions are all possible assignments $ASS^{\mathcal{I}}$.

2.1.4 Determining satisfiability

A constraint is satisfiable iff it has a solution. Not all constraints are satisfiable. For example, the constraint $\mathrm{c}_1 \doteq \mathrm{c}_2$ is unsatisfiable, because for all assignments the denotation of c_1 will be the feature graph $(\mathrm{c}_1, \emptyset)$ and the denotation of c_2 will be $(\mathrm{c}_2, \emptyset)$ which is not the same graph. Clearly a constraint may also define such a clash indirectly, as in the constraint:

$$\begin{array}{l} \mathrm{X}_0 l_1 \doteq \mathrm{X}_1 \\ \mathrm{X}_1 \doteq \mathrm{X}_2 \\ \mathrm{X}_1 l_1 \doteq \mathrm{c}_1 \\ \mathrm{X}_2 l_1 \doteq \mathrm{c}_2 \end{array}$$

The problem whether a constraint is satisfiable is decidable. Algorithms deciding satisfiability for more powerful feature logics (extending the current logic with disjunction and negation) are for example presented in Johnson (1988), Smolka (1989). The present algorithm is an adaptation to $\mathcal{L}$ of the algorithm presented in Smolka (1989). The algorithm consists of a number of simplification rules. Rules are applied until no rules can be applied anymore. In that case the constraint is said to be in normal form or *normal*. For normal constraints it is trivial to check whether the constraint is satisfiable (clash-free). A constraint is *solved* iff it is normal and clash-free.

The simplification algorithm is presented here in two steps. Firstly, I show how to remove all complex paths (paths containing more than one label), by the introduction of some new variables. The resulting constraint, which is called basic, is shown to be satisfiable iff the original constraint was. The next step then rewrites constraints without complex path expressions into *normal form*.

A *basic* constraint is a constraint in which each equation has one of the following forms:

$s \doteq t$
$sl \doteq t$

An arbitrarily constraint ϕ can be mapped into a basic constraint ϕ' by the introduction of new variables. ϕ' is satisfiable iff ϕ is. We will say that two assignments *agree on* a set of variables iff they assign the same elements in the domain to these variables.

Definition 6 (V-equivalence) Two constraints ϕ, ψ are V-equivalent, for V a set of variables, iff:

1. If $\alpha \in \phi^{\mathcal{I}}$, then there exist $\beta \in \psi^{\mathcal{I}}$ such that α and β agree on V.
2. If $\alpha \in \psi^{\mathcal{I}}$, then there exist $\beta \in \phi^{\mathcal{I}}$ such that α and β agree on V.

If constraints ϕ and ψ are V-equivalent then ϕ is satisfiable iff ψ is satisfiable (this follows immediately from the definition).

In the following, C is a constraint, $[X|s]C$ is the constraint obtained from C by replacing each occurrence of variable X with s.

Proposition (Computation of V-equivalent basic constraint). For every constraint ϕ one can compute a V-equivalent basic constraint.

Proof. The following algorithm computes for a given constraint ϕ a basic constraint ϕ'. We will show that ϕ' thus obtained is V-equivalent with ϕ.

Apply any of the rules, until the rules are not applicable:

1. $$\frac{\Gamma,\ s\ l_1 \ldots l_n \doteq d,\ \Delta}{\Gamma,\ s\ l_1 \ldots l_{n-1} \doteq \mathrm{X}_i, \mathrm{X}_i l_n \doteq d,\ \Delta}$$ where $n > 1$, X_i a new variable.
2. $$\frac{\Gamma,\ d \doteq s\ l_1 \ldots l_n\ , \Delta}{\Gamma,\ \mathrm{X}_i \doteq s\ l_1 \ldots l_{n-1}, d \doteq \mathrm{X}_i l_n,\ \Delta}$$ where $n > 1$, X_i a new variable.
3. $$\frac{\Gamma,\ s\ l \doteq t\ l,\ \Delta}{\Gamma,\ s\ l \doteq \mathrm{X}_i, t\ l \doteq \mathrm{X}_i,\ \Delta}$$ where X_i a new variable.

It is easy to verify that if none of the rules is applicable, the resulting constraint is indeed a basic constraint. Also observe that the algorithm terminates because each of the steps replaces an atomic constraint with two new atomic constraints, one of which already is basic, and the other has a shorter path than the previous constraint.

Each step of the algorithm preserves satisfiability, hence a sequence of steps preserves satisfiability as well. Step 1 of the algorithm changes a constraint ϕ into ψ. It is straightforward to show that ϕ is V-equivalent with ψ (and hence ϕ is satisfiable iff ψ is). Assume that $\mathcal{I}$, $\alpha \in \phi^{\mathcal{I}}$ then let β be the assignment which is exactly like α, except that the newly introduced X_i is mapped to the feature graph $\alpha(s)^{l_1 \ldots l_{n-1}}$. Clearly, $\beta \in \psi^{\mathcal{I}}$. The other way around is similar. The same reasoning applies for the steps 2 and 3.

Example 7 Consider the following constraint

$$\phi: \begin{array}{l} \mathrm{X}_0 l_1 l_3 \doteq \mathrm{c}_2 \\ \mathrm{X}_0 l_2 \doteq \mathrm{X}_0 l_1 l_3 \end{array}$$

This constraint is simplified, according to the rules above, in the following three steps, using respectively rule 1, 2 and 3:

$$\phi \Rightarrow \begin{array}{l} \mathrm{X}_0 l_1 \doteq \mathrm{X}_1 \\ \mathrm{X}_1 l_3 \doteq \mathrm{c}_2 \\ \mathrm{X}_0 l_2 \doteq \mathrm{X}_0 l_1 l_3 \end{array} \Rightarrow \begin{array}{l} \mathrm{X}_0 l_1 \doteq \mathrm{X}_1 \\ \mathrm{X}_1 l_3 \doteq \mathrm{c}_2 \\ \mathrm{X}_2 \doteq \mathrm{X}_0 l_1 \\ \mathrm{X}_0 l_2 \doteq \mathrm{X}_2 l_3 \end{array} \Rightarrow \begin{array}{l} \mathrm{X}_0 l_1 \doteq \mathrm{X}_1 \\ \mathrm{X}_1 l_3 \doteq \mathrm{c}_2 \\ \mathrm{X}_2 \doteq \mathrm{X}_0 l_1 \\ \mathrm{X}_2 l_3 \doteq \mathrm{X}_3 \\ \mathrm{X}_0 l_2 \doteq \mathrm{X}_3 \end{array}$$

Given a basic constraint, the following simplification rules rewrite this constraint into its equivalent *normal form*.

Definition 8 (Normal form) Applying any of the following simplification rules to a basic constraint until no rule is applicable, results in a *normal form* constraint:

1. $\dfrac{\Gamma,\ \mathrm{X} \doteq s,\ \Delta}{[X|s]\Gamma,\ \mathrm{X} \doteq s,\ [X|s]\Delta}$ if X occurs in C and $X \neq s$

2. $\dfrac{\Gamma,\ \mathrm{c} \doteq \mathrm{X},\ \Delta}{\Gamma,\ \mathrm{X} \doteq \mathrm{c},\ \Delta}$

3. $\dfrac{\Gamma,\ \mathrm{X}l \doteq s, \mathrm{X}l \doteq t, \Delta}{\Gamma,\ \mathrm{X}l \doteq s, s \doteq t,\ \Delta}$

4. $\dfrac{\Gamma,\ s \doteq s,\ \Delta}{\Gamma, \Delta}$

To show that a normal form of a constraint C computed by this algorithm is equivalent to C observe that each of the simplification rules preserves equivalence. In the first rule the variable X is 'isolated'; in the rest of the constraint the variable is replaced by the constant or variable it is equated with. Clearly in this case ϕ and ψ are equivalent, because, by definition $\alpha(\mathrm{X}) = \alpha(s)$, hence $C^{\mathcal{I}} = C[\mathrm{X}|s]^{\mathcal{I}}$. As for the third rule, note that the solutions of $\{\mathrm{X}l \doteq s, \mathrm{X}l \doteq t\}$ are those assignments such that the descriptors $\mathrm{X}l, s, t$ have the same denotation. The same is the case for the solutions of $\{\mathrm{X}l \doteq s, s \doteq t\}$. Hence the third rule preserves equivalence. The second rule and the fourth rule clearly preserve equivalence.

Furthermore, the simplification algorithm always terminates, because clearly there cannot be an infinite sequence of simplification rules starting from any basic constraint ϕ. To see this, note that there is only a finite number of variables in a given constraint. The first rule 'isolates' such a variable, hence this rule can be applied at most once for each variable; furthermore none of the other rules introduce new variables. The second rule can only be applied a finite number of cases because the number of constants is also finite, and not increased by any of the other rules. The third rule can only be applied a finite number of times because it reduces the length of the paths in a constraint; none of the other rules increase this length. The final rule can only be applied a finite number of times because it reduces the number of equations in a constraint, and none of the other rules increases this number.

The simplification algorithm is very similar to the 'unification' algorithms based on the simplification rules for a system of term equations as presented for example in Apt (1987). Note though that this system does not contain an 'occur check' as I did not exclude cyclic structures.

A normal constraint is clash-free if it does not contain any of the following constraints:

- $\mathrm{c}l \doteq d$ (constant/compound clash)
- $\mathrm{c}_1 \doteq \mathrm{c}_2$ (constant clash)

A normal and clash-free constraint is called *solved*. A solved constraint consists exclusively of atomic constraints of the form:

- $\mathrm{X}l \doteq s$
- $\mathrm{X} \doteq s$

Furthermore, if an equation is of the second type then the variable X occurs only once (the variable is said to be isolated). For this reason it is very easy to see that solved constraints are satisfiable, because they can be interpreted as a recipe to define an appropriate assignment. Such an assignment is called the *principal solution*. For ϕ a solved constraint, $\alpha(x) =_{def} FG[x, \phi]$ is a principal solution of ϕ. The function FG is defined as follows:

$$FG[s,\phi] =_{def} \begin{cases} (s, \emptyset) & \text{if } s \text{ is a constant} \\ FG[t,\phi] & \text{if } s \doteq t \in \phi \\ (s, \{\mathrm{X}lt | \mathrm{X}l \doteq t \in \phi \text{ and } s \rightarrow^*_\phi \mathrm{X}\}) & \text{otherwise} \end{cases}$$

where $\rightarrow^*_\phi$ is the transitive and reflexive closure of $\rightarrow_\phi$ which is a binary relation on the variables occurring in ϕ:

$\mathrm{X} \rightarrow_\phi \mathrm{Y}$ iff $\mathrm{X}l \doteq \mathrm{Y} \in \phi$

Example 9 Consider the following constraint ψ, which was the result of example 7 of the computation of basic constraints:

$$\psi: \begin{array}{l} \mathrm{X}_0 l_1 \doteq \mathrm{X}_1 \\ \mathrm{X}_1 l_3 \doteq \mathrm{c}_2 \\ \mathrm{X}_2 \doteq \mathrm{X}_0 l_1 \\ \mathrm{X}_2 l_3 \doteq \mathrm{X}_3 \\ \mathrm{X}_0 l_2 \doteq \mathrm{X}_3 \end{array}$$

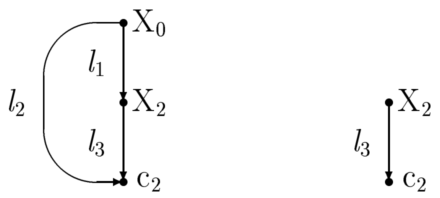
This constraint can then be rewritten into normal form, for example in the following steps, using the rules 3, 1, 3 and 1 of definition 8. The application of the rules 2 and 4 are performed implicitly in the example (for simplicity).

$$\psi \Rightarrow \begin{array}{l} X_0 l_1 \doteq X_1 \\ X_1 \doteq X_2 \\ X_1 l_3 \doteq c_2 \\ X_2 l_3 \doteq X_3 \\ X_0 l_2 \doteq X_3 \end{array} \Rightarrow \begin{array}{l} X_1 \doteq X_2 \\ X_0 l_1 \doteq X_2 \\ X_2 l_3 \doteq c_2 \\ X_2 l_3 \doteq X_3 \\ X_0 l_2 \doteq X_3 \end{array} \Rightarrow \begin{array}{l} X_1 \doteq X_2 \\ X_3 \doteq c_2 \\ X_0 l_1 \doteq X_2 \\ X_2 l_3 \doteq c_2 \\ X_0 l_2 \doteq X_3 \end{array} \Rightarrow \begin{array}{l} X_1 \doteq X_2 \\ X_3 \doteq c_2 \\ X_0 l_1 \doteq X_2 \\ X_2 l_3 \doteq c_2 \\ X_0 l_2 \doteq c_2 \end{array}$$

The principal solution α of this constraint is defined as follows:

$$\begin{array}{l} \alpha(X_0) = (X_0, \{X_0 l_1 X_2, X_2 l_3 X_3, X_0 l_2 X_3\}) \\ \alpha(X_1) = (X_2, \{X_2 l_3 c_2\}) \\ \alpha(X_2) = (X_2, \{X_2 l_3 c_2\}) \\ \alpha(X_3) = (c_2, \emptyset) \\ \alpha(X) = (X, \emptyset) \text{ for } X \notin \{X_0, X_1, X_2, X_3\} \end{array}$$

The first two assignments $\alpha(X_0)$ and $\alpha(X_1)$ can be illustrated as:



Notation.

Once constraints get more complicated they tend to be difficult to read. For that reason I will often use a special representation, called matrix notation, to represent the interpretation of a (satisfiable) constraint on some variable.

The matrix representation of a constraint on a variable is best introduced using an example. For the result of example 9, the matrix representation for the constraints on variable X_0 looks as follows:

$$\begin{bmatrix} l_1 : \begin{bmatrix} l_3 : c_2 \end{bmatrix}_{X_1, X_2} \\ l_2 : c_2 \end{bmatrix}_{X_0}$$

The names of variables only matter in case they are referred to more than once. In the foregoing example, I therefore omit the variables and instead write:

$$\begin{bmatrix} l_1 : \begin{bmatrix} l_3 : c_2 \end{bmatrix} \\ l_2 : c_2 \end{bmatrix}$$

As another example of this notation, consider the following constraint:

X_0 *syn cat* $\doteq$ s
X_0 *sem pred* $\doteq$ visit
X_0 *sem sort* $\doteq$ binary
X_0 *sem arg1 pred* $\doteq$ graham
X_0 *sem arg1 sort* $\doteq$ nullary
X_0 *sem arg2 pred* $\doteq$ haïti
X_0 *sem arg2 sort* $\doteq$ nullary
X_0 *phon in f* $\doteq$ graham
X_0 *phon in r f* $\doteq$ visited
X_0 *phon in r r f* $\doteq$ haïti
X_0 *phon in r r r* $\doteq$ X_0 *phon out*

The matrix representation of the constraints on X_0 looks as follows:

$$\begin{bmatrix} syn : \begin{bmatrix} cat : \mathrm{s} \end{bmatrix} \\ sem : \begin{bmatrix} sort : \mathrm{binary} \\ pred : \mathrm{visit} \\ arg1 : \begin{bmatrix} sort : \mathrm{nullary} \\ pred : \mathrm{graham} \end{bmatrix} \\ arg2 : \begin{bmatrix} sort : \mathrm{nullary} \\ pred : \mathrm{ha\ddot{\i}ti} \end{bmatrix} \end{bmatrix} \\ phon : \begin{bmatrix} in : \begin{bmatrix} f : \mathrm{graham} \\ r : \begin{bmatrix} f : \mathrm{visited} \\ r : \begin{bmatrix} f : \mathrm{ha\ddot{\i}ti} \\ r : []_{X_1} \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ out : []_{X_1} \end{bmatrix} \end{bmatrix}$$

Usually an empty feature structure will not be shown explicitly, but instead only the corresponding variable will be shown, i.e. instead of

$$\begin{bmatrix} r : []_X \end{bmatrix}$$

I write

$$\begin{bmatrix} r : X \end{bmatrix}$$

Furthermore, I use a special notation for parts of such matrices that are used to encode lists and difference lists. If no confusion arises I use the HPSG (Pollard and Sag, 1987) convention of writing a list within angled brackets, where the comma separates elements of the list, and the vertical bar may be used to separate the head from the tail of the list. In path equations the elements of such lists are referred to with attributes *f* and *r* (for first and rest), the empty list is represented with the constant $\langle\rangle$. In case of difference lists I moreover write the 'out' part of the difference list right after the 'in'-part, separated by '-'. The attributes *in* and *out* are used in path equations to refer to these parts. Moreover, in case of a difference list where the tail of the 'in' part is reentrant with the 'out' part, I simply write the 'in' list within " ". As an example I write "letters from mexico get lost" for $\langle letters, from, mexico, get, lost|\mathrm{X}_1\rangle - \mathrm{X}_1$.

As a further abbreviation I sometimes use the functor-argument notation for semantic structures, as introduced in section 1.2.3. Using these abbreviations, the foregoing constraint is written as:

$$\begin{bmatrix} syn : \begin{bmatrix} cat : \mathrm{s} \end{bmatrix} \\ sem : \text{visited(graham,haïti)} \\ phon : \text{"graham visited haïti"} \end{bmatrix}$$

2.2 Adding definite relations

In this section I will apply the construction defined by Höhfeld and Smolka (1988) to the constraint language defined in the preceding section to obtain $\mathcal{R}(\mathcal{L})$. It is shown in Höhfeld and Smolka (1988) how the nice properties of logic programming languages carry over to a whole class of formalisms built on top of arbitrary constraint languages. The idea is to distinguish between the underlying constraint language (for example $\mathcal{E}$, where $\mathcal{E}$ consists of conjunctions of equations between first-order terms) and the definite relations that are defined using constraints from the underlying constraint language. In the case of $\mathcal{E}$ this results in $\mathcal{R}(\mathcal{E})$, which is just first-order logic. From the resulting logic we then take *definite clauses*, and in the case of $\mathcal{E}$ we thus end up with (pure) Prolog. The constraint language $\mathcal{L}$ can be seen as another instance of such a constraint language. To this constraint language I apply the construction sketched above, resulting in $\mathcal{R}(\mathcal{L})$; $\mathcal{R}(\mathcal{L})$ is thus rather similar to pure Prolog, but equations between first order terms are replaced by the path equations introduced in the previous section. Instead of first order terms, our data structures are feature structures. Furthermore, unlike PATR II I will not restrict the formalism by requiring that phrases are built by concatenation. In figure 2.3 it is shown how the different formalisms are related.

2.2.1 Definite clauses of $\mathcal{R}(\mathcal{L})$

I follow the construction defined by Höhfeld and Smolka (1988) to extend $\mathcal{L}$, giving $\mathcal{R}(\mathcal{L})$, by adding a set of relation symbols $\mathcal{R}$, conjunction, negation and existential quantification. The syntax I use for these will be clear from the definition of the interpretations of $\mathcal{R}(\mathcal{L})$ (essentially Prolog syntax where appropriate).

The interpretation $\mathcal{A}$ of $\mathcal{R}(\mathcal{L})$ is defined in terms of the interpretation $\mathcal{I}$ of $\mathcal{L}$. The expression $\alpha[s \leftarrow X]$ defines the assignment which is exactly like α, except possibly for the variable X which gets assigned the element s. An interpretation $\mathcal{A}$ of $\mathcal{R}(\mathcal{L})$ is obtained from $\mathcal{I}$ ($\mathcal{A}$ is said to be based on $\mathcal{I}$) by choosing for every relation symbol $r \in \mathcal{R}$ a relation $r^{\mathcal{A}}$ on $\mathcal{D}^{\mathcal{I}}$ taking the right number of arguments. Furthermore, let

1. $\mathcal{D}^{\mathcal{A}} =_{def} \mathcal{D}^{\mathcal{I}}$;

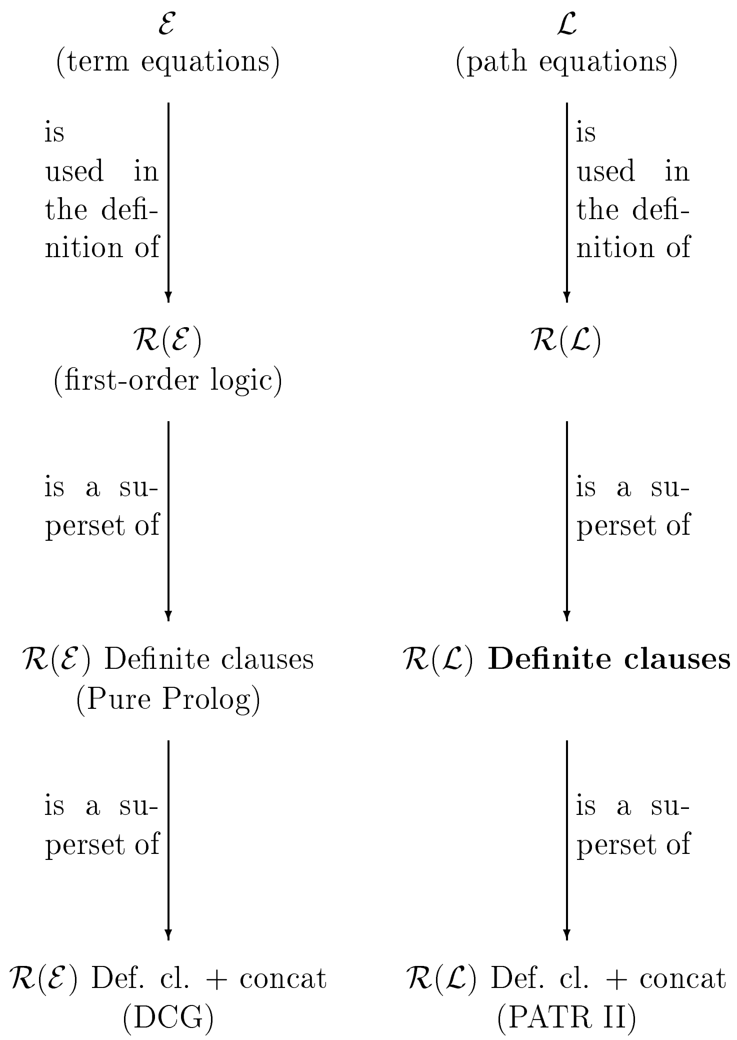


Figure 2.3: Overview of the different formalisms

2. $\phi^{\mathcal{A}} =_{def} \phi^{\mathcal{I}}$; for ϕ a $\mathcal{L}$ constraint.
3. $r(\vec{x})^{\mathcal{A}} =_{def} \{\alpha \in ASS^{\mathcal{A}} | \alpha(\vec{x}) \in r^{\mathcal{A}}\}$;
4. for the empty conjunction $\emptyset$, $\emptyset^{\mathcal{A}} =_{def} ASS^{\mathcal{A}}$;
5. $(F, G)^{\mathcal{A}} =_{def} F^{\mathcal{A}} \cap G^{\mathcal{A}}$;
6. $(\neg F)^{\mathcal{A}} =_{def} ASS^{\mathcal{A}} - F^{\mathcal{A}}$;
7. $(\exists X.F)^{\mathcal{A}} =_{def} \{\alpha[s \leftarrow X] | \alpha \in F^{\mathcal{A}}, s \in \mathcal{D}^{\mathcal{A}}\}$

Implication, universal quantification and disjunction may be defined in terms of these connectives. The formalism consists of definite clauses of $\mathcal{R}(\mathcal{L})$. I write such a definite clause as:

$$p \,{:}\!-\, q_1, \ldots, q_n, \phi.$$

where $p, q_1 \ldots q_n$ are atoms and ϕ is a $\mathcal{L}$ constraint. Atoms look as $r(\mathrm{X}_1, \ldots, \mathrm{X}_n)$ where $r \in R$ and $\mathrm{X}_1 \ldots \mathrm{X}_n \in V$.

A partial order on the set of all $\mathcal{R}(\mathcal{L})$ interpretations is defined as follows: $\mathcal{A} \subseteq \mathcal{B}$ iff for all $r \in R, r^{\mathcal{A}} \subseteq r^{\mathcal{B}}$. The union of a set of $\mathcal{R}(\mathcal{L})$ interpretations is obtained by joining the denotations of the relation symbols and is again an $\mathcal{R}(\mathcal{L})$ interpretation. A model M of a set of definite clauses $\mathcal{S}$ is defined as an $\mathcal{R}(\mathcal{L})$ interpretation such that M satisfies $\mathcal{S}$, i.e., $\mathcal{S}^M = ASS^M$.

The use of definite clauses is motivated by the following theorem, proven in Höhfeld and Smolka (1988) for the general case.

Theorem Let $\mathcal{S}$ be a set of definite clauses in $\mathcal{R}(\mathcal{L})$, and $\mathcal{I}$ a $\mathcal{L}$ interpretation. Define for all $r \in \mathcal{R}$

$$r^{\mathcal{A}_0} =_{def} \emptyset,$$

$$r^{\mathcal{A}_{i+1}} =_{def} \{\alpha | (r(\vec{x}) \,{:}\!-\, G) \in \mathcal{S} \wedge \alpha \in G^{\mathcal{A}_i}\}$$

This defines a chain $\mathcal{A}_0 \subseteq \mathcal{A}_1 \subseteq \ldots$ of $\mathcal{R}(\mathcal{L})$ interpretations, based on $\mathcal{I}$. Moreover, the union

$$\bigcup_{i \geq 0} \mathcal{A}_i$$

is the least model of $\mathcal{S}$ extending $\mathcal{I}$.

This theorem says that if $\mathcal{S}$ is a set of definite clauses, then $\mathcal{S}$ uniquely defines the relations of $\mathcal{R}$; i.e. $\mathcal{S}$ defines unique minimal denotations for the relation symbols of $\mathcal{R}$.

Example 10 As an example, let $C = \{\mathrm{c}\}$, $L = \{l\}$, $V = \{\mathrm{X}\mathrm{X}_1 \ldots\}$ and $R = \{p, q\}$. Furthermore, consider the following definite program:

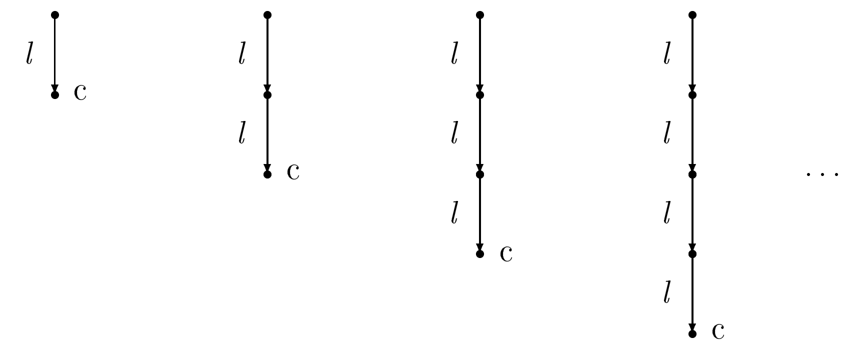


Figure 2.4: Feature graphs $f_1, f_2, f_3 \dots$

(1) $p(\mathrm{X})\mathbin{:\!-}\mathrm{X} \doteq \mathrm{c}.$

$q(\mathrm{X})\mathbin{:\!-}p(\mathrm{X}_1), \mathrm{X} \doteq \mathrm{X}_1.$
$q(\mathrm{X})\mathbin{:\!-}q(\mathrm{X}_1), \mathrm{X}l \doteq \mathrm{X}_1.$

Clearly, $p^{\mathcal{A}_0}$ and $q^{\mathcal{A}_0}$ are both the empty set. Because the assignments are based on $\mathcal{I}$ we know the solutions of the constraints. Therefore, we obtain

$$\begin{aligned} p^{\mathcal{A}_1} &= \{\alpha(\mathrm{X}) | \alpha \in (\mathrm{X} \doteq \mathrm{c})^{\mathcal{A}_0}\} \\ &= \{\alpha(\mathrm{X}) | \alpha \in (\mathrm{X} \doteq \mathrm{c})^{\mathcal{I}}\} \\ &= \{\alpha(\mathrm{X}) | \mathrm{X}_\alpha = \mathrm{c}_\alpha\} \\ &= \{(\mathrm{c}, \emptyset)\} \end{aligned}$$

But the denotation of the relation q is still empty. In the next round, it is clear that $p^{\mathcal{A}_2} = p^{\mathcal{A}_1}$. The denotation of the relation q becomes interesting now:

$$\begin{aligned} q^{\mathcal{A}_2} &= \{\alpha(\mathrm{X}) | \alpha \in (p(\mathrm{Y}), \mathrm{X} \doteq \mathrm{Y})^{\mathcal{A}_1}\} \\ &= \{\alpha(\mathrm{X}) | \alpha \in p(\mathrm{Y})^{\mathcal{A}_1} \cap (\mathrm{X} \doteq \mathrm{Y})^{\mathcal{A}_1}\} \\ &= \{\alpha(\mathrm{X}) | \alpha(\mathrm{Y}) = (\mathrm{c}, \emptyset) \wedge \alpha(\mathrm{X}) = \alpha(\mathrm{Y})\} \\ &= \{(\mathrm{c}, \emptyset)\} \end{aligned}$$

The process continues like that, and we obtain the following diagram. The feature graphs $f_1, f_2, f_3 \dots$ are those given in figure 2.4. The denotation of the relation symbols is given by the following figure, where I write c for the feature graph $(\mathrm{c}, \emptyset)$. [2]

	0	1	2	3	4	5	...
p	$\emptyset$	$\{\mathrm{c}\}$	$\{\mathrm{c}\}$	$\{\mathrm{c}\}$	$\{\mathrm{c}\}$	$\{\mathrm{c}\}$	...
q	$\emptyset$	$\emptyset$	$\{\mathrm{c}\}$	$\{\mathrm{c}, f_1\}$	$\{\mathrm{c}, f_1, f_2\}$	$\{\mathrm{c}, f_1, f_2, f_3\}$	...

A *goal* or *query* is a possibly empty conjunction of $\mathcal{R}(\mathcal{L})$-atoms and a $\mathcal{L}$-constraint, written as:

$?\!-p_1 \dots p_n, \phi.$

[2]Note that in fact the variables in the feature graphs do not matter; hence each of f_i represents a class of feature graphs.

An *answer* to a goal is a satisfiable constraint ψ, such that $\psi \rightarrow p_1 \ldots p_n, \phi$ is valid (given a set of clauses $\mathcal{S}$) in the minimal model of $\mathcal{S}$. For example, for the previous definite clause program a possible answer to the goal

$?\text{-}\, q(\mathrm{X}_0)$

is the answer:

$\mathrm{X}_0 l \doteq \mathrm{c}$

because $\mathrm{X}_0 l \doteq \mathrm{c} \rightarrow q(\mathrm{X}_0)$ is valid in the minimal model.

2.2.2 $\mathcal{R}(\mathcal{L})$-grammars

The formalism defined so far will be used in this thesis in two ways. Firstly I use the formalism to define *grammars* with. However, I also use the formalism to define *meta-interpreters* in. This will become clear in the next section. To separate these two usages, I define a grammar as follows. Without loss of generality I restrict a *grammar G* to consist of definite clauses defining only one unary relation. Restricting grammars to consist of only one relation enables us to distinguish between truly recursive relations and relations that could (at least in principle) be compiled away by partial evaluation techniques. I assume e.g. that the usual templates known from PATR II are already compiled out in such a grammar. As an example of partial evaluation, observe that the following definite clause specification:

(2) $p(\mathrm{X})$:-
    $\mathrm{X} \doteq \langle\rangle$.
  $p(\mathrm{X})$:-
    $\mathrm{X}h \doteq \mathrm{H}$,
    $\mathrm{X}t \doteq \mathrm{T}$,
    $q(\mathrm{H})$
    $p(\mathrm{T})$.

  $q(\mathrm{X})$:-
    $\mathrm{X}l_1 \doteq \mathrm{a}$,
    $\mathrm{X}l_2 \doteq \mathrm{b}$.

can automatically be compiled into the following equivalent specification, where the 'effect' of the predicate $q/1$ is obtained in the predicate $p/1$ directly:

(3) $p(\mathrm{X})$:-
    $\mathrm{X} \doteq \langle\rangle$.
  $p(\mathrm{X})$:-
    $\mathrm{X}h \doteq \mathrm{H}$,
    $\mathrm{X}t \doteq \mathrm{T}$,
    $\mathrm{H}l_1 \doteq \mathrm{a}$,
    $\mathrm{H}l_2 \doteq \mathrm{b}$,
    $p(\mathrm{T})$.

In monolingual grammars, the privileged relation will be called 'sign', as I think of this relation as defining the possible (linguistic) signs.

Note that any set of $\mathcal{R}(\mathcal{L})$-definite clauses can be rewritten as a grammar of $\mathcal{R}(\mathcal{L})$, by 'reification'; for example the following example set of clauses, consisting of several relations

(4) $p(\mathrm{X})$:- ϕ_1.
$p(\mathrm{X})$:- $q(\mathrm{Y}, \mathrm{Z}), \phi_2$.

$q(\mathrm{X}, \mathrm{Y})$:- ϕ_3.
$q(\mathrm{X}, \mathrm{Y})$:- $q(\mathrm{Y}, \mathrm{Z}), \phi_4$.

is rewritten into the following, using the attributes *rel*, *arg1*, *arg2* to represent the relation symbol and the arguments:

(5) *sign*(P) :-
  P *rel* $\doteq$ p1,
  P *arg1* $\doteq$ X, ϕ_1.
sign(P) :-
  sign(Q),
  P *rel* $\doteq$ p1,
  P *arg1* $\doteq$ X,
  Q *rel* $\doteq$ q2,
  Q *arg1* $\doteq$ Y,
  Q *arg2* $\doteq$ Z, ϕ_2.
sign(Q) :-
  Q *rel* $\doteq$ q2,
  Q *arg1* $\doteq$ X,
  Q *arg2* $\doteq$ Y, ϕ_3.
sign(Q) :-
  sign(Q),
  Q *rel* $\doteq$ q2,
  Q *arg1* $\doteq$ X, ϕ_4.

Before I continue to define the procedural semantics of $\mathcal{R}(\mathcal{L})$ in the next section, I will first introduce some notational conveniences as follows. Using the matrix representation introduced above, I often leave out the constraints in a definite clause, and instead replace the variables in the clause with the matrix notation of the equations constraining that variable. Moreover, if the predicate symbol of an atom is *sign*/1 then I sometimes leave the predicate symbol out; hence

$sign(\mathrm{X}_0)$:- $sign(\mathrm{X}_1), \ldots sign(\mathrm{X}_n), \phi$.

may be written simply as:

M_0 :- $\mathrm{M}_1 \ldots \mathrm{M}_n$.

where M_i are the matrices defining the constraints on X_i. For example, the rule

$$
\begin{array}{l}
sign(\mathrm{X}_0)\,\texttt{:-} \\
\quad sign(\mathrm{X}_1), \\
\quad sign(\mathrm{X}_2), \\
\quad \mathrm{X}_0\ syn\ cat \doteq \mathrm{s}, \\
\quad \mathrm{X}_1\ syn\ cat \doteq \mathrm{np}, \\
\quad \mathrm{X}_2\ syn\ cat \doteq \mathrm{vp}, \\
\quad \mathrm{X}_1\ syn\ agr \doteq \mathrm{X}_2\ syn\ agr.
\end{array}
$$

is written:

$$
\begin{array}{l}
sign(\left[\ syn : \left[\ cat : \mathrm{s}\ \right]\ \right])\,\texttt{:-} \\
\quad sign(\left[\ syn : \left[\begin{array}{l} cat : \mathrm{np} \\ agr : \mathrm{Agr} \end{array}\right]\ \right]), \\
\quad sign(\left[\ syn : \left[\begin{array}{l} cat : \mathrm{vp} \\ agr : \mathrm{Agr} \end{array}\right]\ \right]).
\end{array}
$$

2.3 Procedural Semantics

In this section I define the procedural semantics of our formalism using the standard terminology as for example presented in Kowalski (1979). The procedural semantics for $\mathcal{R}(\mathcal{L})$ will be similar to Prolog's procedural semantics. For grammars I will investigate different procedural semantics, i.e. parsing and generation strategies, in the chapters 3 and 4. However, as I will define these alternative strategies as meta interpreters within $\mathcal{R}(\mathcal{L})$ itself, it will still be useful to define the 'basic' procedural semantics here.

2.3.1 Solving a query

Recall that a *goal* or *query* Q is a possibly empty conjunction of $\mathcal{R}(\mathcal{L})$-atoms and a $\mathcal{L}$-constraint, written as:

$$\texttt{?-}p_1 \ldots p_n, \phi.$$

The purpose of a proof procedure is to show whether, for a given set of definite clauses $\mathcal{S}$, there is a satisfiable constraint ψ, such that $\psi \rightarrow Q$ is valid. The proof procedure is a refutation procedure, which shows that the denial of some goal is inconsistent with the assumptions $\mathcal{S}$. The building blocks of proof procedures are *inference rules.* I will assume a *top-down* inference strategy. The initial goal is replaced by other goals via inference rules. If the empty goal is obtained, then a refutation has been discovered (where an empty goal is a goal without any atoms). The constraint associated with the empty goal is the desired answer.

In later chapters I will define proof procedures in which the inference rules are applied *bottom-up.* In such a proof procedure new assertions are derived from the

to_lend(B) :-
    own(B),
    $available$(B).

own(B) :-
    B $author \doteq$ greene,
    B $title \doteq$ 'the captain and the enemy',
    B $status \doteq$ in.
own(B) :-
    B $author \doteq$ greene,
    B $title \doteq$ 'travels with my aunt',
    B $status \doteq$ out.
own(B) :-
    B $author \doteq$ melville,
    B $title \doteq$ 'moby dick',
    B $status \doteq$ in.

$available$(B) :-
    B $status \doteq in$.

Figure 2.5: A definite clause specification of the state of an hypothetical library.

assumptions $\mathcal{S}$ until an assertion is derived that explicitly contradicts the denial of the goal.

The top-down refutation procedure is best introduced using an example. Consider the set of clauses in figure 2.5 which define the state of a hypothetical library. In this database it is asserted that all books can be lent which are owned by the library and which are available. Furthermore, the library owns three books, which are described by its author, title and status. A book is available if its 'status' is 'in'. Suppose we want to know which books by 'greene' can be lent according to our database. In that case the goal is defined as:

?- to_lend(Book),
    Book $author \doteq$ greene.

which should be read as 'for which Book, Book's author is greene and Book can be lent'. However, in the refutation procedure such a goal will be denied, i.e. read as 'for all Book it is not the case that Book's author is greene and Book can be lent'. The inference rule in a top-down refutation procedure is a generalization of 'modus tollens'. From the fact that 'owning' plus 'availability' implies 'lendability', and from the goal 'Book cannot be lent' we can infer that it can neither be the case that Book is both owned and available, i.e. our original goal is replaced by:

?- *own*(Book),
available(Book),
Book *author* $\doteq$ greene.

Next we can argue as follows. If it is the case that Book is not both owned and available, and we know that a book with the title 'the captain and the enemy', and status 'in' *is* owned, then we infer that such a book is not available:

?- *available*(Book),
Book *author* $\doteq$ greene,
Book *title* $\doteq$ 'the captain and the enemy',
Book *status* $\doteq$ in.

Finally, from this goal (reading: a book with title 'the captain and the enemy', author 'greene' and status 'in' is not available) we can infer the empty clause because it is stated in the database that all things with status 'in' in fact *are* available:

?- Book *author* $=$ greene,
Book *title* $\doteq$ 'the captain and the enemy',
Book *status* $\doteq$ in.

Hence we obtain a contradiction, thus the negation of the original goal turns out not to be true. Therefore the original goal is shown to be true. Furthermore we obtain a constraint on Book that can be viewed as a counter example to the negated goal and hence constitutes an answer to the goal.

The proof procedure sketched here will now be defined as follows. Note that I use the standard terminology as for example presented in Kowalski (1979). First I define the *inference rule* more precisely. Then I will define the *computation* rule which tells us *on which atom* the inference rule must be applied. Finally I define the *search* rule which tells us *against which clause* the inference rule has to be applied.

The inference rule I will define is called *goal-reduction*. The inference rule will select an atom from the goal and will replace this atom with the body of a clause defining this atom. Furthermore, the constraint associated with the new goal will be the conjunction of the constraints ϕ and ψ, where ϕ is the constraint associated with the old goal and ψ is the constraint associated with the body of the selected clause. Summarizing, for a goal $p_1 \ldots p_n, \phi$ goal reduction will replace one of the atoms p_i in the goal by the atoms in the body of a clause defining p_i as follows:

$$p_1 \ldots p_{i-1} p_i(\mathrm{X}_1 \ldots \mathrm{X}_n) p_{i+1} \ldots p_n, \phi \Rightarrow_{\mathcal{S}} p_1 \ldots p_{i-1}, q_1 \ldots q_n, p_{i+1} \ldots p_n, \phi, \psi$$

where

$$p_i(\mathrm{X}_1 \ldots \mathrm{X}_n) \texttt{:-} q_1 \ldots q_n, \psi$$

is a variant of a clause in $\mathcal{S}$, such that the variables in the clause do not occur in the original goal, except for the variables explicitly mentioned. As usual, a variant of a clause is obtained by consistently renaming its variables. For example, given the clauses in the library example above, consider the goal:

$$\begin{array}{ll} \texttt{?-} & own(\mathrm{X}_3), \\ & available(\mathrm{X}_3), \\ & \mathrm{X}_3\ author \doteq \mathrm{melville}. \end{array}$$

Suppose we want to try to apply the inference rule on the atom $own(\mathrm{X}_3)$ against the third clause defining the predicate 'own/1'. In order for $own(\mathrm{X}_3)$ and $own(\mathrm{B})$ to match, we first rename the variables in the clause, obtaining the variant:

$$\begin{array}{l} own(\mathrm{X}_3) \texttt{:-} \\ \quad \mathrm{X}_3\ author \doteq \mathrm{melville}, \\ \quad \mathrm{X}_3\ title \doteq \text{`moby dick'}, \\ \quad \mathrm{X}_3\ status \doteq \mathrm{in}. \end{array}$$

and next the body of this clause replaces the selected atom in the goal, obtaining:

$$\begin{array}{ll} \texttt{?-} & available(\mathrm{X}_3), \\ & \mathrm{X}_3\ author \doteq \mathrm{melville}, \\ & \mathrm{X}_3\ title \doteq \text{`moby dick'}, \\ & \mathrm{X}_3\ status \doteq \mathrm{in}. \end{array}$$

The inference rule is applied only in case the resulting constraint ϕ, ψ is satisfiable, because clearly if this constraint is not satisfiable then we will not be able to produce a correct answer; a constraint remains unsatisfiable whatever information is added to it (i.e. whatever constraints we conjoin it with). Therefore each time the inference rule is applied, it is tested whether the resulting constraint is still satisfiable. This satisfiability test will use the procedure described in the previous section. Therefore, the result of an inference rule will either be a goal in which the $\mathcal{L}$-constraint is solved, or either be 'failure' if the current part of the search space does not contain any solutions. The operation to test whether the conjunction of two constraints is satisfiable relates to the 'unification' operation in Prolog and PATR II, or to 'constraint-solving' in other formalisms.

Note that it is not strictly necessary to compute after each reduction step whether the resulting constraint is satisfiable (as long as proposed answers are checked to be satisfiable). This observation gives rise to inference procedures where satisfiability checks are 'delayed' or performed less frequently. This may be useful if the check costs a lot of overhead in comparison to the size of the search space that could be avoided. For an application of this idea in a linguistic setting, consider the work on Constraint Logic Grammars (Damas and Varile, 1990).

The *computation rule* of the proof procedure determines which atom in a given goal is to be reduced. As in Prolog, I assume that the leftmost atom of a goal is

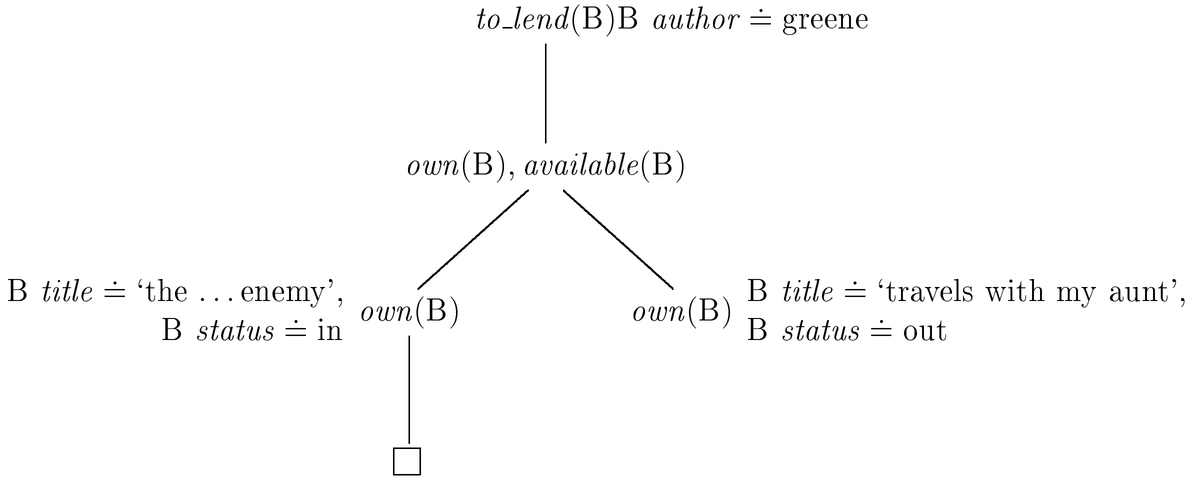


Figure 2.6: The search tree for the query whether it is possible to lend a book by Greene. A local tree corresponds to all possible ways in which the leftmost atom of the mother node can be reduced.

always reduced; i.e. the computation rule selects the leftmost atom in a goal. In later chapters I will argue that for parsing and generation a computation rule which selects the *head* may be much more suitable. [3]

Finally, the *search rule* of a proof procedure determines in which order the search space is traversed. For a given selected atom several clauses may be available to reduce the atom with. Each of the possible reductions may lead to the desired result; hence this defines a search space which has the shape of a tree. For our toy example above, this tree looks as in figure 2.6, assuming the leftmost selection rule. As another example, suppose we always select the *rightmost* atom of a goal. Clearly this results in the same answer; however the search tree looks different, as is clear from the search tree for the same example in figure 2.7, but now with the rightmost selection rule. In this case, the search space is minimal and refutation proceeds deterministically, i.e. there is no backtracking. The thing to note here is that a different computation rule might have important effects on the size of the search space. This fact will be exploited in later chapters, where more efficient proof procedures for parsing and generation are investigated by employing a linguistically motivated computation rule.

[3] Some confusion can easily arise here. In the logic programming tradition, the head of a definite clause $p \text{:-} q_1 \ldots q_n$ is the atom p; in the linguistic tradition the head of a rule is one of the daughters of the rule. Viewing a rule as a definite clause, as I do here, thus gives rise to two notions of head: the 'linguistic' head is one of q_i, whereas the 'logical' head is p. In using the term 'head' I will refer to the linguistic notion of 'head'.

$$\mathit{to\_lend}(\text{Book})\ \text{Book}\ \mathit{author} \doteq \text{greene}$$

$$\mathit{available}(\text{Book}), \mathit{in}(\text{Book})$$

$$\mathit{available}(\text{Book})\ \text{Book}\ \mathit{status} \doteq \text{in}$$

$$\square\ \text{Book}\ \mathit{title} \doteq \text{`the \ldots enemy'}$$

Figure 2.7: The search tree of the query 'which book by Greene can be lent'. In this case, each local tree represents all possible ways in which the rightmost atom of the mother node can be reduced. As the tree does not branch, the search proceeds deterministically.

I assume, as in Prolog, that search trees are traversed in a depth-first left-to-right order. Backtracking occurs if branches are encountered which contain no refutation. Note that the search-rule is quite independent of both the direction of the proof procedure (bottom-up, top-down) and the computation rule. Apart from the usual depth-first strategy I adopt here it is often possible to augment proof procedures with well-formed and ill-formed subgoal tables. In section 4.6.5 I will come back to this subject.

At this point it may be useful to introduce another type of tree, which we may call 'parse tree'. In such a tree a node represents an atom in a goal; the daughters of a node are the atoms with which this atom is reduced. A leaf in the parse tree thus represents a goal of which the reduction yielded the empty goal. [4] A search tree thus represents all possible refutations; a parse-tree represents only one refutation. In examples of parse trees I usually replace the variables in the atom with the constraints imposed on this variable (in matrix representation); also I leave out the predicate symbol in the case of the predicate symbol *sign*/1. For example, for the goal "which books by 'greene' can be lent?" we have the following parse tree, leaving out the constraints on the variable Book.

[4] It is thus assumed that the start goal only contains one atom.

$$sign(\begin{bmatrix} cat : \mathrm{s} \\ phon : \mathrm{P}_0 - \mathrm{P} \end{bmatrix})\texttt{:-}$$

$$\quad sign(\begin{bmatrix} cat : \mathrm{np} \\ phon : \mathrm{P}_0 - \mathrm{P}_1 \end{bmatrix}), sign(\begin{bmatrix} cat : \mathrm{vp} \\ phon : \mathrm{P}_1 - \mathrm{P} \end{bmatrix}).$$

$$sign(\begin{bmatrix} cat : \mathrm{vp} \\ phon : \mathrm{P}_0 - \mathrm{P} \end{bmatrix})\texttt{:-}$$

$$\quad sign(\begin{bmatrix} cat : \mathrm{tv} \\ phon : \mathrm{P}_0 - \mathrm{P}_1 \end{bmatrix}), sign(\begin{bmatrix} cat : \mathrm{np} \\ phon : \mathrm{P}_1 - \mathrm{P} \end{bmatrix}).$$

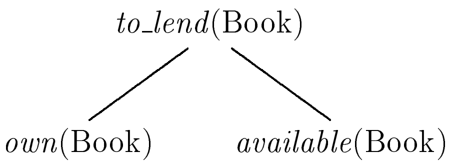
$$sign(\begin{bmatrix} cat : \mathrm{np} \\ phon : \langle \mathrm{greene} | \mathrm{T} \rangle - \mathrm{T} \end{bmatrix}.)$$

$$sign(\begin{bmatrix} cat : \mathrm{np} \\ phon : \langle 21, \mathrm{stories} | \mathrm{T} \rangle - \mathrm{T} \end{bmatrix}).$$

$$sign(\begin{bmatrix} cat : \mathrm{vp} \\ phon : \langle \mathrm{slept} | \mathrm{T} \rangle - \mathrm{T} \end{bmatrix}).$$

$$sign(\begin{bmatrix} cat : \mathrm{tv} \\ phon : \langle \mathrm{wrote} | \mathrm{T} \rangle - \mathrm{T} \end{bmatrix}).$$

Figure 2.8: This grammar of $\mathcal{R}(\mathcal{L})$ encodes a simple context-free grammar, by a difference list implementation of concatenation.



Example 11 (context free grammars) As an example of a correct refutation according to the refutation procedure defined above, consider the encoding of a simple context free grammar in figure 2.8 as a grammar of $\mathcal{R}(\mathcal{L})$.

The following goal is solved as in figure 2.9; the corresponding parse tree is presented in figure 2.10. Note that in the trace of the refutation I use the up-arrow to indicate that the constraints on the variable are identical to the constraints on that variable in the preceding goal.

$$\texttt{?-}sign(\begin{bmatrix} phon : \langle \mathrm{greene}, \mathrm{wrote}, 21, \mathrm{stories} \rangle - \langle\rangle \end{bmatrix}_{\mathrm{X}_0}).$$

$$
\begin{array}{lll}
 & \texttt{?-} & sign(\mathrm{X}_0), \\
 & & \left[\ phon : \langle \text{greene}, \text{wrote}, 21, \text{stories} \rangle - \langle\rangle\ \right]_{\mathrm{X}_0}. \\
\Rightarrow & \texttt{?-} & sign(\mathrm{X}_1), sign(\mathrm{X}_2), \\
 & & \left[\begin{array}{l} cat : \text{s} \\ phon : \langle \text{greene}, \text{wrote}, 21, \text{stories} \rangle - \langle\rangle \end{array}\right]_{\mathrm{X}_0}, \\
 & & \left[\begin{array}{l} cat : \text{np} \\ phon : \langle \text{greene}, \text{wrote}, 21, \text{stories} \rangle - \mathrm{P} \end{array}\right]_{\mathrm{X}_1}, \\
 & & \left[\begin{array}{l} cat : \text{vp} \\ phon : \mathrm{P} - \langle\rangle \end{array}\right]_{\mathrm{X}_2}. \\
\Rightarrow & \texttt{?-} & sign(\mathrm{X}_2), \mathrm{X}_0 \Uparrow, \\
 & & \left[\begin{array}{l} cat : \text{np} \\ phon : \langle \text{greene}, \text{wrote}, 21, \text{stories} \rangle - \langle \text{wrote}, 21, \text{stories} \rangle \end{array}\right]_{\mathrm{X}_1}, \\
 & & \left[\begin{array}{l} cat : \text{vp} \\ phon : \langle \text{wrote}, 21, \text{stories} \rangle - \langle\rangle \end{array}\right]_{\mathrm{X}_2}. \\
\Rightarrow & \texttt{?-} & sign(\mathrm{X}_3), sign(\mathrm{X}_4), \mathrm{X}_0 \Uparrow, \mathrm{X}_1 \Uparrow, \mathrm{X}_2 \Uparrow, \\
 & & \left[\begin{array}{l} syn : \text{tv} \\ phon : \langle \text{wrote}, 21, \text{stories} \rangle - \mathrm{P}_1 \end{array}\right]_{\mathrm{X}_3}, \\
 & & \left[\begin{array}{l} cat : \text{np} \\ phon : \mathrm{P}_1 - \langle\rangle \end{array}\right]_{\mathrm{X}_4}. \\
\Rightarrow & \texttt{?-} & sign(\mathrm{X}_4), \mathrm{X}_0 \Uparrow, \mathrm{X}_1 \Uparrow, \mathrm{X}_2 \Uparrow, \\
 & & \left[\begin{array}{l} syn : \text{tv} \\ phon : \langle \text{wrote}, 21, \text{stories} \rangle - \langle 21, \text{stories} \rangle \end{array}\right]_{\mathrm{X}_3}, \\
 & & \left[\begin{array}{l} cat : \text{np} \\ phon : \langle 21, \text{stories} \rangle - \langle\rangle \end{array}\right]_{\mathrm{X}_4}. \\
\Rightarrow & \texttt{?-} & \mathrm{X}_0 \Uparrow, \mathrm{X}_1 \Uparrow, \mathrm{X}_2 \Uparrow, \mathrm{X}_3 \Uparrow, \mathrm{X}_4 \Uparrow .
\end{array}
$$

Figure 2.9: A trace of the refutation of the claim that "greene wrote 21 stories" is *not* a string in the grammar. That is, a trace of the proof that this string in fact is recognized by the grammar.

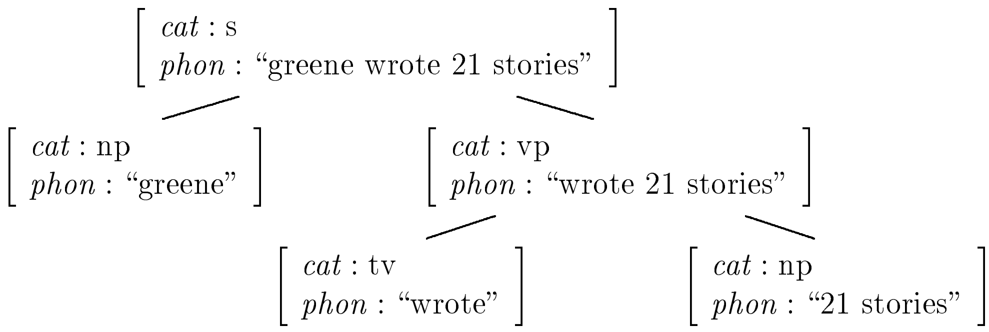


Figure 2.10: Parse tree of the string "greene wrote 21 stories"

2.3.2 Meta-interpreter

Note that I did restrict a grammar to consist only of a set of definite clauses defining the predicate *sign*/1. However, I will also use definite clauses of $\mathcal{R}(\mathcal{L})$ to define meta-interpreters for such grammars. For these meta-interpreters I will assume the operational semantics sketched above. In such cases I treat the definite clauses defining *sign* as data; a clause:

$$\mathit{sign}(\mathrm{X}_0)\,\texttt{:-}\,\mathit{sign}(\mathrm{X}_1)\ldots\mathit{sign}(\mathrm{X}_n),\phi.$$

is represented as the clause

$$\mathit{rule}(\mathrm{X}_0,\langle \mathrm{X}_1,\ldots \mathrm{X}_n\rangle)\,\texttt{:-}\,\phi.$$

i.e. the body of the clause is represented as a list. A meta-interpreter for $\mathcal{R}(\mathcal{L})$ thus may be defined as in program 6, which follows the procedural semantics of $\mathcal{R}(\mathcal{L})$, i.e. it is a top-down proof procedure with a left-most computation rule and a depth-first search rule.

(6) *refutation*(Goal) :-
        rule(Goal, Ds),
        refutations(Ds).

    refutations(⟨⟩).
    refutations(⟨H|T⟩) :-
        refutation(H),
        refutations(T).

For example, a rightmost computation rule can be implemented by defining the predicate *refutations* as follows:

(7) *refutations*(⟨⟩).

$$\begin{array}{l} \mathit{refutations}(\langle \mathrm{H}|\mathrm{T}\rangle)\texttt{:-} \\ \quad \mathit{refutations}(\mathrm{T}), \\ \quad \mathit{refutation}(\mathrm{H}). \end{array}$$

2.4 Parsing and generation

2.4.1 Unrestricted parsing problem

A grammar G as defined above is a definite clause specification of the relation $sign/1$. Such a grammar defines what possible linguistic objects are. Each of these objects, or signs, will be specified among several dimensions. For example, a grammar for English may define that the sentence 'Mexican priests drink whisky' is a sign of English, i.e. one of the answers to the goal

$\texttt{?-}\mathit{sign}(\mathrm{S}).$

may be something like the following:

$$\begin{bmatrix} \mathit{syn} : \mathrm{s} \\ \mathit{sem} : \text{drink([mexican](priests),whisky)} \\ \mathit{phon} : \text{"mexican priests drink whisky"} \end{bmatrix}_{\mathrm{S}}$$

Given a grammar, the parsing problem consists of a specification of a string. Parsing then enumerates the signs which have this string as their value of the attribute *phon*. A generation problem consists of a specification of a semantic representation. A generator then enumerates the signs which have this semantic representation as the value of their *sem* attribute. Instead of the paths *phon* and *sem* we may of course use any other paths. Both the parsing problem and the generation problem can be defined as a goal. For example, the problem to parse 'priests drink mexican whisky' may be defined as the following goal:

$$\texttt{?-} \quad \begin{bmatrix} \mathit{phon} : \text{"priests drink mexican whisky"} \end{bmatrix}$$

Similarly, the generation problem, defined by a number of $\mathcal{L}$ constraints, for example looks as:

$$\texttt{?-} \quad \begin{bmatrix} \mathit{sem} : \text{drink(priests,[mexican](whisky))} \end{bmatrix}$$

It is of course possible to add other constraints such as the syntactic category of the signs we are interested in. A typical example of a generation goal is:

$$\texttt{?-} \quad \begin{bmatrix} \mathit{cat} : \mathrm{s} \\ \mathit{subcat} : \langle\rangle \\ \mathit{sem} : \text{see(graham,drink([mexican](priest),[strong](whisky)))} \end{bmatrix}$$

Therefore, the *unrestricted parsing problem* consists of a grammar G and the goal

$$\texttt{?-} \quad \mathit{sign}(\mathrm{X}_0), \phi.$$

The answer to the unrestricted parsing problem is an answer to this goal with respect to the grammar. This notion is introduced to compare it with several 'restricted' versions later in this section.

2.4.2 Problems with the unrestricted parsing problem

Defining parsing- and generation problems as goals implies that the parser will enumerate all feature structures that have a *compatible* value for the *phon* attribute; and similarly such a generator will enumerate feature structures that have a compatible value for the *sem* attribute. This approach faces a number of problems however. For example, consider the generation problem for a grammar that defines among others the following signs:

$$\left[\begin{array}{ll} \textit{phon}: & \text{“the priest drinks”} \\ \textit{sem}: & \left[\begin{array}{l} \textit{pred}: \text{drink} \\ \textit{arg1}: \left[\ \textit{pred}: \text{priest}\ \right] \end{array}\right] \end{array}\right]$$

$$\left[\begin{array}{ll} \textit{phon}: & \text{“the priest drinks whisky”} \\ \textit{sem}: & \left[\begin{array}{l} \textit{pred}: \text{drink} \\ \textit{arg1}: \left[\ \textit{pred}: \text{priest}\ \right] \\ \textit{arg2}: \left[\ \textit{pred}: \text{whisky}\ \right] \end{array}\right] \end{array}\right]$$

For a given logical form

$$\left[\begin{array}{l} \textit{pred}: \text{drink} \\ \textit{arg1}: \left[\ \textit{pred}: \text{priest}\ \right] \end{array}\right]$$

the generator delivers the strings “the priest drinks” and “the priest drinks whisky”, and perhaps also “the priest drinks strong cheap whisky from a brown paper bag”. On the other hand, the generator also delivers “the priest drinks” for the logical form [5]

$$\left[\begin{array}{l} \textit{pred}: \text{drink} \\ \textit{arg1}: \left[\ \textit{pred}: \text{priest}\ \right] \\ \textit{arg2}: \left[\ \textit{pred}: \text{whisky}\ \right] \end{array}\right]$$

A related problem can be illustrated with respect to parsers. For example, in parsers for Definite Clause Grammars (Pereira and Warren, 1980) strings are (usually) represented by difference lists; for example ‘the priest drinks from a brown paper bag’ is represented as:

```
[the,priest,drinks,from,a,brown,paper,bag|X]-X
```

However, the parser usually does not expect to find an input goal such as:

```
?- s([the,priest,drinks,from,a,brown,paper,bag|X]-X
```

[5] Note that this particular instantiation of the problem partly disappears using the *sort* labels as introduced in the previous section to differentiate between semantic structures of different arity.

but instead expects the tail variable of the difference list to be instantiated with some constant (often the empty list: `[]`), eg:

```
?- s([the,priest,drinks,from,a,brown,paper,bag]-[]).
```

This 'trick' simplifies the parser considerably because it is now impossible to further instantiate the tail variable. Most DCG parsers will not terminate without this convention because these parsers will try to further instantiate the variable tail by longer and longer lists of words.

2.4.3 A restricted version of the parsing problem

It seems, then, that what we really intend with the generation problem is something like: 'show me all signs that place exactly the following constraints on the following semantic representation'. In Wedekind (1988) this is formalized for generation with LFG's by requiring that the input structure subsumes (is more general, is less informative) the structure that is generated (*completeness*), and moreover, the structure that is generated subsumes the input structure (*coherence*). [6] In Wedekind's proposal no distinction is made between different kinds of information. That is, the generator is not allowed to add any syntactic, morphological and semantic information. However, in the case of eg. generation, it seems reasonable to require completeness and coherence only for the value of the path that is used to represent the semantic representation. The generator should be allowed to add all kind of syntactic information, and most notably of course the value of the *phon* attribute itself! Similarly, for parsing I require completeness and coherence of the attribute representing the string.

Therefore, it is useful to be able to refer to the meaning of a certain constraint ϕ restricted to a path p. I define the *restriction* of a constraint ϕ with respect to path p, written ϕ/p to be

$$(\phi/p)^{\mathcal{I}} =_{def} \{\beta \in ASS^{\mathcal{I}} | \exists\alpha \in \phi^{\mathcal{I}} \text{ such that } p^{\mathcal{I}}_{\alpha} = p^{\mathcal{I}}_{\beta}\}$$

Furthermore, I introduce the notion p-parsing problem, that generalizes over parsing and generation (and transfer, cf. chapter 5), where p is some (fixed) path. The idea is that for something to be a proper answer to the goal it must be the case that the constraints on the path p proposed by that answer are equivalent to the constraints on path p that were already present in the formulation of the goal. In the next definition of the parsing problem we require that for some path p the constraint in the goal restricted to p is equivalent to the answer restricted to p.

The parsing problem restricted to a path p, called the p-parsing problem is defined as follows. Note that the path p is fixed.

[6] Note that this usage of the terms completeness and coherence should not be confused with the usual meaning of these terms in logic. Neither should it be confused with yet another notion of completeness and coherence which is used in LFG to enforce subcategorization requirements.

Definition 12 (p-Parsing Problem) A *p-parsing problem* consists of a grammar G and a goal q:

?-$sign(\mathrm{X}), \phi$.

The answer to a p-parsing problem is a solved constraint ψ such that

- ψ is an answer to q with respect to G; and
- $(\phi/\mathrm{X}p)^{\mathcal{I}} = (\psi/\mathrm{X}p)^{\mathcal{I}}$

Returning to the usual implementation of parsers for DCG, it turns out that instantiating the *out*-variable of the difference list with the empty string in fact implements one part of this extra equivalence condition (the 'coherence' part in Wedekind's terminology); as in DCG (i.e. Prolog) it is possible to implement a coherence check by 'freezing' all variables (replacing them with fresh constants, eg. by the `numbervars` predicate) occurring in the original goal.

Coherence can be implemented in the current framework by a similar technique. It is possible to add constraints to the input structure instantiating all parts that are not mentioned in the constraint on the path p to constants $\mathrm{c}_1 \ldots \mathrm{c}_k$ that are not used otherwise in the grammar. This will block any further instantiation of the value at p, and hence implements coherence. Completeness can be implemented, as discussed in Shieber *et al.* (1990), by maintaining a distinction in the derivation between the constraints added by the grammar and the constraints stemming from the original goal. At the end of the derivation it is then possible to compare the two constraints to see whether they are indeed equivalent. Wedekind (1988) discusses the implementation of completeness and coherence for LFG's.

I will assume in the following that the p-parsing problem is defined as in definition 12. However, in the meta interpreters for $\mathcal{R}(\mathcal{L})$-grammars that are defined in the next chapters I abstract away from the implementation of completeness and coherence, for didactic reasons. This move is justifiable because the implementation is rather straightforward and quite independent of the different inference strategies to be discussed. Therefore, leaving completeness and coherence out clarifies the differences between the different strategies. On the other hand, I continue to assume that the parsing problem is defined as above (with completeness and coherence) in order to be able to discuss certain termination properties.

In the remainder of this thesis I assume the previous definition of the p-parsing problem as in 12. For some applications it may be useful to consider other definitions. Some possibilities are discussed shortly.

2.4.4 Other versions of the parsing problem

Other definitions of the parsing and generation problems have been defined as well. For example, in van Noord (1990b) two relaxations of the foregoing definition of the p-parsing problem are discussed. Such relaxations may in certain applications allow for simpler grammars.

Restricting equivalence to cyclic labels. The first relaxation assumes that it is possible to make a distinction between *cyclic* and *non-cyclic* labels. A non-cyclic label will be a label with a finite number of possible values (i.e. it is not recursive). For example the labels *arg1* and *arg2* may be cyclic whereas the label *number* may be non-cyclic. The completeness and coherence condition can be restricted to the values of cyclic labels. If the proof procedure can only further instantiate acyclic labels no termination problems occur because there are only a finite number of possibilities to do this.

For certain applications this may be useful. For example, consider the case where monolingual grammars define semantic structures which are annotated with some syntactic information as well. If the completeness and coherence conditions are restricted to cyclic labels, the input to the generator may be under-specified with respect to these syntactic decorations. These syntactic labels can then be filled in by the generator on the basis of the monolingual grammar.

No equivalence for reentrancies. The second relaxation has to do with reentrancies in feature structures. It is possible to define a version of the parsing problem that does not take into account such reentrancies. As will be explained in more detail in section 5.4.2, it turned out that in using $\mathcal{R}(\mathcal{L})$ to define transfer rules in an MT system it was rather cumbersome to be forced to redefine possible reentrancies in transfer rules as they were defined in the monolingual grammar. Therefore, a definition of the p-parsing problem was investigated that did not require completeness and coherence of such reentrancies. The possible usefulness of this conception of the parsing problem will be discussed in section 5.4.2.

Thompson's proposal. Another possibility is investigated in Thompson (1991). The basic intuition of his approach is that the parser (or generator) should come up with those signs that are as close as possible to the input structure. That is, answers to the parsing and generation problem consist of those signs that 'minimally extend' the input and 'maximally overlap' the input. The notions 'minimally extend' and 'maximally overlap' are defined *with respect to other possible answers* to the parsing problem.

The problem with this approach seems to be that, although interesting, the implementation is far from straightforward. The difficulty is increased by the fact that in this approach for the proof procedure to know whether something is an answer to a goal it is necessary to take into account all other possible answers. In the other versions of the parsing problem answers are independent of each other.

2.5 Post Correspondence problem

In this section I show that the p-parsing problem for $\mathcal{R}(\mathcal{L})$-grammars is generally not solvable. A yes-no problem is *undecidable* (cf. Hopcroft and Ullman (1979), pp.178-

179) if there is no algorithm that takes as its input an *instance* of the problem and determines whether the answer to that instance is 'yes' or 'no'. An instance of a problem consists of a particular choice of the *parameters* of that problem.

In the case at hand I show that in general the p-parsing problem is not solvable for any fixed path p. I show that this result holds both for the p-parsing problem, and goals in general. I encode an undecidable problem in a $\mathcal{R}(\mathcal{L})$-grammar in such a way that deciding whether there is at least one solution of the p-parsing problem will be equivalent to solving this undecidable problem.

I use Post's Correspondence Problem (PCP) as the undecidable problem. The following definition and example of a PCP are taken from Hopcroft and Ullman (1979)[chapter 8.5].

An instance of PCP consists of two lists, $A = v_1 \ldots v_k$ and $B = w_1 \ldots w_k$ of strings over some alphabet Σ. This instance has *a solution* if there is any sequence of integers $i_1 \ldots i_m$, with $m \geq 1$, such that

$$v_{i_1}, v_{i_2}, \ldots, v_{i_m} = w_{i_1}, w_{i_2}, \ldots, w_{i_m}.$$

The sequence $i_1, \ldots, i_m$ is a solution to this instance of PCP. As an example, assume that $\Sigma = \{0, 1\}$. Furthermore, let $A = \langle 1, 10111, 10 \rangle$ and $B = \langle 111, 10, 0 \rangle$. A solution to this instance of PCP is the sequence 2,1,1,3 (obtaining the sequence 101111110). For an illustration, cf. figure 2.11.

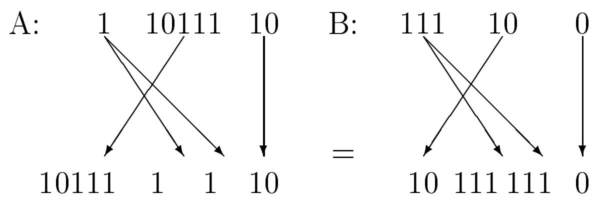


Figure 2.11: Illustration of a solution of a PCP problem.

Clearly there are PCP's that do not have a solution. Assume again that $\Sigma = \{0, 1\}$. Furthermore let $A = \langle 1 \rangle$ and $B = \langle 0 \rangle$. Clearly this PCP does not have a solution. In general, however, the problem whether some PCP has a solution or not is not decidable. This result is proved by Hopcroft and Ullman (1979) by showing that the halting problem for Turing Machines can be encoded as an instance of Post's Correspondence Problem.

First I give a simple algorithm to encode any instance of a PCP as a grammar of $\mathcal{R}(\mathcal{L})$, in such a way that the question whether there is a solution to this PCP can be phrased as a *goal*. Then I extend the encoding in such a way that the question whether there is a solution is equivalent to the p-parsing problem.

Note that I use the notation l^i to stand for i repetitive occurrences of label l. Hence the expression X_0 *a in* r^4 *f* will stand for the path X_0 *a in r r r r f*. Furthermore I assume that C and L are defined appropriately.

Encoding of PCP.

$$sign(\left[\begin{array}{l} a : \mathrm{A}_1 - \mathrm{A} \\ b : \mathrm{B}_1 - \mathrm{B} \end{array}\right]) \texttt{:-} sign(\left[\begin{array}{l} a : \mathrm{A}_1 - \mathrm{A}_2 \\ b : \mathrm{B}_1 - \mathrm{B}_2 \end{array}\right]), sign(\left[\begin{array}{l} a : \mathrm{A}_2 - \mathrm{A} \\ b : \mathrm{B}_2 - \mathrm{B} \end{array}\right]).$$

$$sign(\left[\begin{array}{l} a : \langle 1|\mathrm{X}\rangle - \mathrm{X} \\ b : \langle 1,1,1|\mathrm{Y}\rangle - \mathrm{Y} \end{array}\right]).$$

$$sign(\left[\begin{array}{l} a : \langle 1,0,1,1,1|\mathrm{X}\rangle - \mathrm{X} \\ b : \langle 1,0|\mathrm{Y}\rangle - \mathrm{Y} \end{array}\right]).$$

$$sign(\left[\begin{array}{l} a : \langle 1,0|\mathrm{X}\rangle - \mathrm{X} \\ b : \langle 0|\mathrm{Y}\rangle - \mathrm{Y} \end{array}\right]).$$

Figure 2.12: The grammar of $\mathcal{R}(\mathcal{L})$ corresponding to example PCP

1. For each $1 \leq i \leq k$ (k the length of lists A and B) define a unit rule $sign(\mathrm{X}_0) \texttt{:-} \phi$, where ϕ is defined as follows (the $i-th$ member of A is $a_1 \ldots a_m$, and the i-th member of B is $b_1 \ldots b_n$).
   (a) For each $j, 1 \leq j \leq m$ there is an equation: $\mathrm{X}_0\ a\ in\ r^{j-1}\ f \doteq \mathrm{a}_j$
   (b) Moreover, there is an equation: $\mathrm{X}_0\ a\ in\ r^m \doteq \mathrm{X}_0\ a\ out$.
   (c) For each $j, 1 \leq j \leq n$ there is an equation: $\mathrm{X}_0\ b\ in\ r^{j-1}\ f \doteq \mathrm{b}_j$
   (d) Moreover, there is an equation: $\mathrm{X}_0\ b\ in\ r^n \doteq \mathrm{X}_0\ b\ out$.
2. There is one non unit rule, defined as follows:

$$(8)\quad sign(\left[\begin{array}{l} a : \mathrm{A}_1 - \mathrm{A} \\ b : \mathrm{B}_1 - \mathrm{B} \end{array}\right]) \texttt{:-} sign(\left[\begin{array}{l} a : \mathrm{A}_1 - \mathrm{A}_2 \\ b : \mathrm{B}_1 - \mathrm{B}_2 \end{array}\right]), sign(\left[\begin{array}{l} a : \mathrm{A}_2 - \mathrm{A} \\ b : \mathrm{B}_2 - \mathrm{B} \end{array}\right]).$$

The underlying idea of the algorithm is really very simple. For each pair of strings from the lists A and B there will be one unit rule where these strings are represented by a difference-list encoding. Furthermore there is a general combination rule that simply concatenates A-strings and concatenates B-strings.

The following goal then is equivalent to determining whether there is a solution to the PCP:

$$\texttt{?-}\quad \left[\begin{array}{l} a : \mathrm{L} - \langle\rangle \\ b : \mathrm{L} - \langle\rangle \end{array}\right].$$

In matrix representation the resulting $\mathcal{R}(\mathcal{L})$ grammar for the first example PCP above, look as in figure 2.12. Furthermore, one of the parse trees for the solution given above is presented in figure 2.13.

To show that the same result applies to the p-parsing problem I change the encoding slightly. Firstly, each unit rule built by the foregoing algorithm will have an extra constraint $\mathrm{X}_0\ solution \doteq \mathrm{no}$. Similarly, in the combination rule I add for each of the

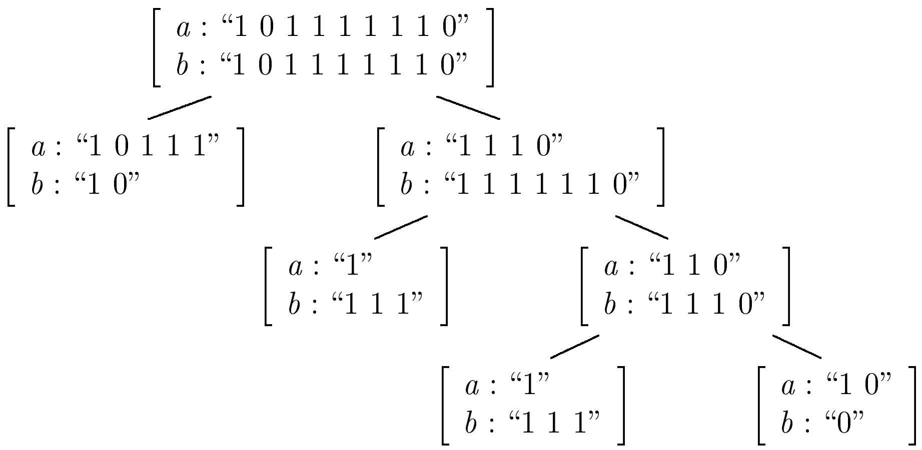


Figure 2.13: Parse tree of the proof that there is a solution, according to the $\mathcal{R}(\mathcal{L})$ encoding of the example PCP.

three variables $\mathrm{X}_0, \mathrm{X}_1, \mathrm{X}_2$ the constraints X_i *solution* $\doteq$ no. Finally, the query above is then encoded as an extra rule as follows:

$$(9)\quad \begin{bmatrix} \textit{solution} : \text{yes} \end{bmatrix} \texttt{:-} \begin{bmatrix} \textit{solution} : \text{no} \\ a : \mathrm{L} - \langle\rangle \\ b : \mathrm{L} - \langle\rangle \end{bmatrix}.$$

Now, the question whether there is a solution to an instance of PCP can be answered "yes" in case the enumeration of the *solution*-parsing problem of the following formula at least yields one answer:

?- *sign*(Sign),
Sign *solution* $\doteq$ yes.

Proposition The p-parsing problem for $\mathcal{R}(\mathcal{L})$-grammars is unsolvable.

Proof. Suppose the problem *was* solvable. In that case we could use it to solve the PCP, because a PCP π has a solution if and only if for its encoding, $\mathcal{R}(\mathcal{L})(\pi)$, the query

$$\texttt{?-}\ \begin{bmatrix} \textit{solution} : \text{yes} \end{bmatrix}$$

has at least one solution. By construction, there is a direct relation to a solution to π (a list of integers) and a derivation of $\mathcal{R}(\mathcal{L})(\pi)$. A derivation encodes the concatenations of the a and b strings. If and only if such a concatenation yields the same list this derivation can be the daughter of the final rule. The last problem however is known to be undecidable, hence the p-parsing problem is unsolvable.

2.6 Conclusion

Constraint-based grammars can be used, in principle, both for parsing and generation. This is also true for grammars defined in $\mathcal{R}(\mathcal{L})$. However, to use such grammars in a practically interesting way, the grammars need to be restricted in some way, as the parsing and generation problem of $\mathcal{R}(\mathcal{L})$ grammars is undecidable. Historically, the restriction has been (in order for parsing to be efficient), that phrases are built by concatenation. No restriction for generation was assumed, as generation played a minor role. But, generation on the basis of declarative grammars is not without problems. For example, the simple top-down, left-to-right backtrack search strategy leads to non-termination for linguistically motivated grammars. Chapter 3 discusses several (linguistically relevant) problems for some obvious generation techniques. Furthermore, that chapter provides motivation for a different processing strategy, in which generation proceeds essentially *bottom-up*, and *head-driven*. A simple version of this strategy is defined, and its properties and short-comings are investigated. Several variants and possible improvements are discussed. Relying on the notion 'head' has some implications for the way in which semantic structures should be combined in the grammar. Essentially, the semantic-head-driven generation algorithm embodies the assumption that semantic structures are defined in a *lexical* and *head-driven* fashion. It turns out that the head-driven generation strategy faces problems with analyses in which this assumption is violated. As an important example, the semantic-head-driven generation strategy faces problems, if the head of a construction has been 'displaced'. Such an analysis is often assumed for verb-second phenomena in for example German and Dutch, if the grammars are restricted to be concatenative. Thus, the assumption that semantic structures are built lexically and head-driven, does not make linguistic sense, if phrases are to be built by concatenation. In order for these assumptions to make linguistic sense, it is therefore necessary to have more freedom in the way phonological structures are combined. Thus, instead of concatenative grammars, non-concatenative grammars are called for.

In chapter 4 linguistic motivation for such non-concatenative operations on phonological representations is discussed. A number of proposals for operations like 'head-wrapping' and 'sequence-union' are described, and a class of formalisms is introduced in which operations on strings are allowed which are *linear* and *non-erasing*. Formalisms such as Head-Grammars (Pollard, 1984) and Tree Adjoining Grammars (Joshi *et al.*, 1975), are members of this class. Clearly, parsing algorithms developed for concatenative formalisms cannot be used for these more powerful formalisms. An important question thus is how to parse with non-concatenative grammars. I describe a very general algorithm for this class of grammars which proceeds, again, *head-driven*. The motivation for such a processing strategy is discussed. Furthermore, I describe possible extensions and improvements of the basic algorithm. I also show how the parser can be 'specialized' for lexicalized, and constraint-based, versions of Tree Adjoining

Grammars.

In order for grammars to be effectively used for both parsing and generation, such grammars need to adhere to the assumptions that, on the one hand, semantic structures are built in a lexical and head-driven fashion, and on the other hand, that phonological structures are built in a *linear* and *non-erasing* way.

Chapter 3

Semantic-head-driven Bottom-up Generation

3.1 Introduction

Natural language generation often is characterized as a process in which the following two subprocesses can be identified. The *conceptual* part of this process decides *what* should be said in a given situation. The conceptual part thus constructs a 'message', i.e. some sort of semantic representation. The other, *grammatical*, part of natural language generation then takes as its input such a semantic representation and decides *how* this meaning representation can be realized linguistically. For a discussion of this division of labor see for example Appelt (1987), and also McKeown (1985); Thompson (1977); the distinction is also sometimes characterized in terms of a *strategical* and a *tactical* part.

The interest in this thesis will be in the second, i.e. grammatical (tactical), part of natural language generation. Thus, I will invariably assume that the conceptual part (or 'planner') provides appropriate semantic structures — our task will be to realize these semantic structures in natural language. A grammar defines the relation between semantic structures and phonological structures. Therefore, the task of generation is to compute this relation, given its definition in the form of a grammar.

Almost any modern linguistic theory assumes, that a natural language grammar not only describe the correct sentences of a language, but that such a grammar also describes the corresponding semantic structures of the grammatical sentences. Given that a grammar specifies the relation between phonology and semantics it seems obvious that the generator is supposed to use this specification. For example, for Generalized Phrase Structure Grammars (GPSG) Gazdar *et al.* (1985)[chapters 9 and 10] provide a detailed description of the semantic interpretation of the sentences licensed by the grammar. In my view, a generator based on GPSG should construct a sentence for a given semantic structure, according to the semantic interpretation rules of GPSG. However, Busemann (1990) presents a generator, which is said to be based on GPSG, but which does not take as its input, as one would expect, a logical form, but rather some kind of control expression which merely instructs the grammatical component which rules of the grammar to apply. Similarly, in the conception of Gardent and Plainfossé (1990), the generator is provided with some kind of 'deep structure' which can be interpreted as a control expression instructing the grammar which rules to apply. These approaches to the generation problem clearly 'solve' some of the problems encountered in generation — simply by pushing the problem away into the conceptual component. The generation problem I discuss in this chapter is somewhat more involved. I assume that the relation between semantic structures and phonological structures is defined by a declarative (constraint-based) grammar. The task of a generator is to compute, for a given semantic structure, the corresponding phonological structures, according to such a grammar.

As I will discuss in the next section, several approaches to the (grammatical) generation problem for constraint-based grammars are not completely satisfactory. As an alternative I propose a simple semantic-head-driven bottom-up generator (called BUG) which, as I will show, is superior in some respects to these others. [1] Furthermore I discuss some problems of BUG and some extensions to BUG. I also clarify the relationship of BUG to generators of the same family, such as the generators presented in Calder *et al.* (1989), and Shieber *et al.* (1990).

I argue that the head-driven bottom-up approach should be favored because in this approach the order of processing is geared towards the input (head-driven) and the information available in lexical entries (bottom-up).

3.2 A simple grammar for Dutch

To illustrate the different generation techniques, which I discuss in the following sections, I will first define a simple, but in some respects typical, grammar for a small subset of Dutch.

Concatenation. I assume in this grammar that all strings are built using a difference-list implementation of concatenation as in concatenative formalisms. Therefore, each binary rule extends the following:

$$sign(\left[\; phon : \mathrm{P}_0 - \mathrm{P} \;\right})\texttt{:-}sign(\left[\; phon : \mathrm{P}_0 - \mathrm{P}_1 \;\right]), sign(\left[\; phon : \mathrm{P}_1 - \mathrm{P} \;\right]).$$

To make the rules somewhat easier to read, I will not explicitly mention these constraints in the rules — however for each rule they are present.

Lexical entries will generally specify their phonology as follows, where the variable Word is instantiated by some constant representing the terminal symbol associated with that lexical entry:

$$sign(\left[\; phon : \langle \mathrm{Word} | \mathrm{Tail} \rangle - \mathrm{Tail} \;\right]).$$

Note though that none of the conclusions of this chapter in any way depends on the restriction to assume such a concatenative base. In the next chapter I discuss other

[1] This generator was originally defined in van Noord (1989).

ways to combine strings — the generation algorithms discussed here are all capable of handling such more powerful rules. In fact, some of the problems I will encounter for generation, can be solved in grammars in which concatenation is not the sole operation to construct phonological structures.

Subcategorization lists, and a lexical, head-driven construction of semantic structures. In the grammar it is assumed that verb phrases are built using a subcategorization list as in HPSG (Pollard and Sag, 1987). Elements of the subcat list are selected one at the time by a binary verb phrase rule (the use of the attributes *v2* and *lex* will be explained later):

$$
(1)\quad \textit{sign}(\begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sc} : \text{Tail} \\ \textit{sem} : \text{Sem} \\ \textit{lex} : \text{no} \\ \textit{v2} : \text{Verb2} \end{bmatrix}) :\!-
$$

$$
\textit{sign}(\text{Arg}),
$$

$$
\textit{sign}(\begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sc} : \langle \text{Arg}|\text{Tail} \rangle \\ \textit{sem} : \text{Sem} \\ \textit{v2} : \text{Verb2} \end{bmatrix}).
$$

The value of the subcat feature (the label *sc*) is a list of signs. In this rule, the first element of the subcat list of the second daughter of the rule, is equated with the first daughter of the rule. The 'remaining' elements on the list, i.e. the tail of the list is 'percolated' to the mother node of the rule. If a verb selects several arguments then this vp rule can be applied iteratively. The following example clarifies this technique. Assume that some verb subcategorizes for four elements, called *a, b, c* and *d*. Then the parse tree for the saturated verb phrase dominating this verb, looks as in figure 3.1. Thus, the elements of the subcat list are selected one at the time. Note that in the case where elements are selected to the left of the head, the order of the elements on the subcat list is the reverse of the order of the actual elements in the string. Furthermore note that if a sign is saturated, then its subcat list is empty.

Furthermore, it is stated in this rule, that the semantics of the second daughter is identical with the semantics of the mother of the node. In the current grammar, semantic structures will invariably be built lexically; these structures are always unified between the semantic-head and the mother of a rule. Thus, the semantic-head, or 'functor', of a rule is that daughter, which shares its semantics with the semantics of the mother node of the rule. This daughter not necessarily is the 'syntactic-head' of the phrase. For example, modifiers often are analyzed as the semantic-head of the construction they modify, whereas the modified part of the construction is the syntactic head.

The semantics of a lexical entry is defined by sharings with the semantics of the elements it subcategorizes for. Some verbs are defined as in rule 2. In these entries it should be noted how semantic structures are defined by sharings with parts of the

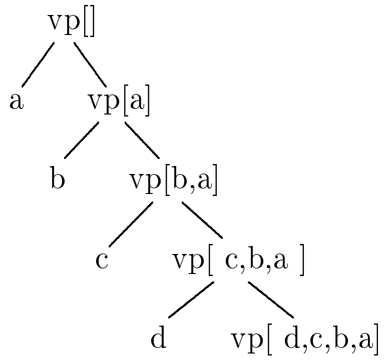


Figure 3.1: This figure illustrates the use of subcategorization lists. The elements of this list are selected by a binary verb-phrase rule, one at the time. If selection is to the left, then the order of the elements on the list mirrors the order of the elements found in the string.

elements on the subcat list. Therefore, if such verbs are selected by the VP rule above, the semantics is gradually instantiated, when the arguments are selected. Note that this mechanism is essentially the mechanism assumed in UCG (Zeevat *et al.*, 1987); see also Moore (1989), and Nerbonne (1992) for discussion.

$$(2)\quad sign(\begin{bmatrix} cat: \text{vp} \\ sem: \text{slaapt(Exp)} \\ lex: \text{yes} \\ v2: \text{no\_v2} \\ sc: \langle \begin{bmatrix} cat: \text{np} \\ sem: \text{Exp} \end{bmatrix} \rangle \\ phon: \text{“slaapt”} \end{bmatrix}).$$

$$(3)\quad sign(\begin{bmatrix} cat: \text{vp} \\ sem: \text{vertelt(Ag,Th)} \\ lex: \text{yes} \\ v2: \text{no\_v2} \\ sc: \langle \begin{bmatrix} cat: \text{np} \\ sem: \text{Th} \end{bmatrix}, \begin{bmatrix} cat: \text{np} \\ sem: \text{Ag} \end{bmatrix} \rangle \\ phon: \text{“vertelt”} \end{bmatrix}).$$

Verb second. In the previous analysis of subcategorization, the arguments of a verb always precede the verb, which is correct in Dutch if we do not take into account verb second phenomena. I will now indicate how verb second phenomena can be accounted for.

In Dutch, the finite verb occupies the second position of a main clause, whereas in subordinate clauses it occupies the final position. Thus we have:

(4) a. Jan berekent de kosten door
Jan computes the costs through
Jan computes the costs

b. omdat Jan de kosten doorberekent
because Jan the costs through computes
because Jan computes the costs

In order to be able to use the same verb phrase rules in both subordinate and main clauses, I will define a threading implementation of a 'movement' analysis of verb second. This analysis uses the features *v2* and *lex*, already mentioned in the foregoing rule 1. I assume that in main clauses the finite verb also occupies the final position, but in a phonologically empty way. Furthermore the information of this empty verb is then percolated through the *v2* feature to the pre-VP position. The basic idea of this analysis is illustrated in figure 3.2. Furthermore, categories of type q (for 'question'

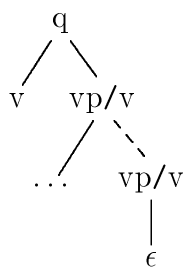


Figure 3.2: The analysis of verb second in Dutch. The information of the initial finite verb is percolated downwards to a phonologically empty verb, in the position of the finite verb in subordinate sentences.

— the rest of the root sentence has a yes-no question word-ordering) consist of a finite verb and a saturated verb phrase *that misses this verb*. This is how the current grammar deals with verb second. The rule is defined in 5.

(5)
$$
\begin{array}{l}
sign(\begin{bmatrix} cat:\mathrm{q} \\ sem:\mathrm{Sem} \end{bmatrix}) \mathrel{:-} \\
\quad sign(\begin{bmatrix} cat:\mathrm{vp} \\ lex:\mathrm{yes} \end{bmatrix}_{\mathrm{Verb2}}), \\
\quad sign(\begin{bmatrix} cat:\mathrm{vp} \\ sc:\langle\rangle \\ sem:\mathrm{Sem} \\ v2:\mathrm{Verb2} \end{bmatrix}).
\end{array}
$$

In this rule the binary feature *lex* is used to implement the fact that only verbs, and not verb phrases, can be fronted to the verb-second position. The information of the verb in verb second position is percolated through the *v2* feature. Furthermore, there is the option that a verb in Dutch can be 'empty', in case the features in its 'incoming' *v2* feature unify with its own features, cf. rule 6.

$$(6)\quad sign(\begin{bmatrix} cat : \mathrm{vp} \\ sem : \mathrm{Sem} \\ sc : \mathrm{Sc} \\ lex : \mathrm{no} \\ v2 : \begin{bmatrix} cat : \mathrm{vp} \\ sem : \mathrm{Sem} \\ sc : \mathrm{Sc} \end{bmatrix} \\ phon : \mathrm{P} - \mathrm{P} \end{bmatrix}).$$

In case a verb phrase should not dominate this empty verb, the grammar instantiates the v2 feature with some constant, for example the value no_v2. The grammar rule in rule 7 defines that a complementizer phrase consists of a complementizer and the argument for which this complementizer subcategorizes.

$$\begin{aligned} (7)\quad & sign(\begin{bmatrix} sem : \mathrm{Sem} \\ cat : \mathrm{comp} \\ sc : \langle\rangle \end{bmatrix}) \text{:-} \\ & \quad sign(\begin{bmatrix} sem : \mathrm{Sem} \\ cat : \mathrm{comp} \\ sc : \langle \mathrm{Arg} \rangle \end{bmatrix}), \\ & \quad sign(\mathrm{Arg}). \end{aligned}$$

Such a complementizer 'omdat' (the Dutch equivalent of 'because') is defined in rule 8, where it should be noted that this complementizer requires that the verb phrase should not dominate the empty verb, by specifying the value of the *v2* attribute. Furthermore note that the complementizer requires that the embedded verb phrase should be 'saturated', i.e. should have selected its arguments, because it requires that the value of the *sc* attribute of the verb phrase for which it subcategorizes, is the empty list. This is a way to implement LFG's *completeness* requirement (Bresnan, 1982) on subcategorization specifications. In general, lexical entries require that the subcat lists of their arguments are empty.

$$(8)\quad sign(\begin{bmatrix} cat : \mathrm{comp} \\ sem : \mathrm{omdat(Sem)} \\ sc : \langle \begin{bmatrix} cat : \mathrm{vp} \\ sem : \mathrm{Sem} \\ sc : \langle\rangle \\ v2 : \mathrm{no\_v2} \end{bmatrix} \rangle \\ phon : \text{“omdat”} \end{bmatrix}).$$

Modification. The current grammar specifies that a Dutch root sentence may consist of an adverbial phrase, followed by a node of the special category 'q'. Rule 9 defines this option.

$$(9)\quad sign(\begin{bmatrix} cat : \text{root} \\ sem : \text{Sem} \end{bmatrix}) \text{:-} \\ sign(\begin{bmatrix} cat : \text{adv} \\ sc : \langle \text{Q} \rangle \\ sem : \text{Sem} \end{bmatrix}), \\ sign(\begin{bmatrix} cat : \text{q} \end{bmatrix}_{\text{Q}}).$$

Note that the position of this adverbial is usually analyzed as the topic position. The current simplification is motivated, because for the expository purposes of the grammar, it is not necessary to implement a gap-threading analysis of topicalization.

The grammar also allows some simple modification of verb phrases. Verb phrases may consist of an adverbial phrase followed by a verb phrase. The subcat list of the verb phrases is percolated, because in Dutch, unlike in English, adverbials can be interspersed with the arguments of the verbs:

(10) a. dat Jan Arie de leugens vandaag vertelt
that John Arie the lies today tells
that John tells the lies to Arie today

b. dat Jan Arie vandaag de leugens vertelt

c. dat Jan vandaag Arie de leugens vertelt

Note that such sentences motivate the use of the binary verb phrase rule 1, giving rise to branching verb phrases, rather than flat verb phrases as in HPSG. Rule 11 defines that a verb-phrase may consist of an adverbial and a verb phrase.

$$(11)\quad sign(\begin{bmatrix} cat : \text{vp} \\ sem : \text{Sem} \\ sc : \text{Sc} \\ lex : \text{no} \\ v2 : \text{Verb2} \end{bmatrix}) \text{:-} \\ sign(\begin{bmatrix} cat : \text{adv} \\ sem : \text{Sem} \\ sc : \langle \text{Vp} \rangle \end{bmatrix}), \\ sign(\begin{bmatrix} cat : \text{vp} \\ sc : \text{Sc} \\ v2 : \text{Verb2} \end{bmatrix}_{\text{Vp}}).$$

Such an adverbial might for example be defined such as the following entry of 'vandaag' (today) in rule 12.

$$(12)\quad sign(\begin{bmatrix} cat : \mathrm{adv} \\ sem : [\mathrm{vandaag}](\mathrm{E}) \\ sc : \langle \begin{bmatrix} sem : \mathrm{E} \end{bmatrix} \rangle \\ phon : \text{“vandaag”} \end{bmatrix}).$$

Idioms Certain idiomatic constructions can be defined, simply by a further requirement on the elements the head of the idiomatic construction subcategorizes for. For example, the idiomatic construction ‘een dutje doet’ (to take a nap), is analyzed in such a way that the verb ‘doet’ simply selects a noun-phrase of which the semantics is restricted to be ‘dutje’. Furthermore, the semantics of the verb is a one-place predicate, comparable to ‘sleep’.

$$(13)\quad sign(\begin{bmatrix} cat : \mathrm{vp} \\ sem : \mathrm{dutje\_doet}(\mathrm{Exp}) \\ lex : \mathrm{yes} \\ v2 : \mathrm{no\_v2} \\ sc : \langle \begin{bmatrix} cat : \mathrm{np} \\ sem : \mathrm{Exp} \end{bmatrix}, \begin{bmatrix} cat : \mathrm{np} \\ sem : dutje \end{bmatrix} \rangle \\ phon : \text{“doet”} \end{bmatrix}).$$

We will not be very much interested in noun phrases; therefore I simply assume some noun phrases that are defined as in the following example for ‘Arie’, ‘een dutje’ (‘a nap’) and ‘leugens’ (‘lies’):

$$(14)\quad sign(\begin{bmatrix} cat : \mathrm{np} \\ sem : \mathrm{arie} \\ phon : \text{“Arie”} \end{bmatrix}).$$

$$(15)\quad sign(\begin{bmatrix} cat : \mathrm{np} \\ sem : \mathrm{dutje} \\ phon : \text{“een dutje”} \end{bmatrix}).$$

$$(16)\quad sign(\begin{bmatrix} cat : \mathrm{np} \\ sem : \mathrm{leugens} \\ phon : \text{“leugens”} \end{bmatrix}).$$

This second noun phrase will be used to construct the idiomatic verb phrase ‘een dutje doen’ which means ‘to take a nap’.

A suitable parser for this simple grammar returns for the call

$$\texttt{?-}\quad sign(\begin{bmatrix} phon : \text{“omdat arie leugens vertelt”} \end{bmatrix}_{\mathrm{X}_0}).$$

the following constraint on X_0:

$$\begin{bmatrix} \mathit{cat} : \mathit{comp} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \text{omdat(vertelt(arie,leugens))} \\ \mathit{phon} : \text{“omdat arie leugens vertelt”} \end{bmatrix}_{X_0}$$

The resulting grammar is given in the figures 3.3, 3.4 and 3.5. Note that I left out the relation symbols ‘sign’ for short.

3.3 Problems with existing approaches

This section gives an overview of some of the problems, that some previous algorithms for grammatical generation have. I discuss the LFG generator of Wedekind (1988), the DCG generator of Dymetman and Isabelle (1988) and Shieber's chart-based generator (Shieber, 1988). Firstly I show that top-down generators have problems with (linguistically relevant) examples of ‘left recursion’. On the other hand, the bottom-up generator of Shieber requires grammars to be ‘semantically monotonic’, a notion that I explain below. In the next section I show how BUG handles grammars that are not semantically monotonic.

3.3.1 Top-down generators and left recursion

Consider the ‘naive’ top-down generation algorithm defined in chapter 2 and repeated here as figure 3.6. The algorithm simply matches a node with the mother node of some rule and recursively generates the daughters of this rule. The left-to-right search strategy we have been assuming, causes non-termination of this algorithm for the goal:

$$(17)\ \ ?\text{-}\mathit{refutation}(\begin{bmatrix} \mathit{cat} : \text{comp} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \text{omdat(slaapt(arie))} \\ \mathit{v2} : \text{no\_v2} \end{bmatrix}).$$

The task is to generate a saturated complementizer phrase. The binary complementizer rule 7 is the only candidate and may be applied. After the proper generation of the first daughter of this rule the generator attempts to generate a saturated verb phrase with semantics slaapt(arie). For this embedded generation task the verb phrase rule 1 is the appropriate candidate. The first daughter of this rule, however, is not sufficiently instantiated to guide the search any further: the algorithm may for example apply the same rule 1 for this node, and will go into an infinite loop because each time it chooses rule 1 for the first daughter of rule 1.

Therefore, the order in which nonterminals are expanded is very important, as was noticed by Dymetman and Isabelle (1988) and Wedekind (1988). If, in the foregoing example, the generator first tries to generate the second daughter, then it may be possible to generate the first daughter without problems afterwards. In the approach

```
% vp -> xp, vp
```

$$\begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{sc}: \mathrm{Tail} \\ \mathit{sem}: \mathrm{Sem} \\ \mathit{lex}: \mathit{no} \\ \mathit{v2}: \mathrm{V} \end{bmatrix} \texttt{:-}\mathrm{Arg}, \begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{sc}: \langle \mathrm{Arg} | \mathrm{Tail} \rangle \\ \mathit{sem}: \mathrm{Sem} \\ \mathit{v2}: \mathrm{V} \end{bmatrix}. \tag{1}$$

```
% root -> adv, q
```

$$\begin{bmatrix} \mathit{cat}: \mathrm{root} \\ \mathit{sem}: \mathrm{Sem} \end{bmatrix} \texttt{:-} \begin{bmatrix} \mathit{cat}: \mathrm{adv} \\ \mathit{sc}: \langle \mathrm{Q} \rangle \\ \mathit{sem}: \mathrm{Sem} \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \mathrm{q} \end{bmatrix}_{\mathrm{Q}}. \tag{9}$$

```
% q -> v2, vp
```

$$\begin{bmatrix} \mathit{cat}: \mathrm{q} \\ \mathit{sem}: \mathrm{Sem} \end{bmatrix} \texttt{:-} \begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{lex}: \mathrm{yes} \end{bmatrix}_{\mathrm{V}}, \begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{sc}: \langle\rangle \\ \mathit{sem}: \mathrm{Sem} \\ \mathit{v2}: \mathrm{V} \end{bmatrix}. \tag{5}$$

```
% cp -> comp, xp
```

$$\begin{bmatrix} \mathit{sem}: \mathrm{Sem} \\ \mathit{cat}: \mathrm{comp} \\ \mathit{sc}: \langle\rangle \end{bmatrix} \texttt{:-} \begin{bmatrix} \mathit{sem}: \mathrm{Sem} \\ \mathit{cat}: \mathrm{comp} \\ \mathit{sc}: \langle \mathrm{Arg} \rangle \end{bmatrix}, \mathrm{Arg}. \tag{7}$$

```
% vp -> adv, vp
```

$$\begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{sem}: \mathrm{Sem} \\ \mathit{sc}: \mathrm{Sc} \\ \mathit{lex}: \mathrm{no} \\ \mathit{v2}: \mathrm{V} \end{bmatrix} \texttt{:-} \begin{bmatrix} \mathit{cat}: \mathrm{adv} \\ \mathit{sem}: \mathrm{Sem} \\ \mathit{sc}: \langle \mathrm{Vp} \rangle \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \mathrm{vp} \\ \mathit{sc}: \mathrm{Sc} \\ \mathit{v2}: \mathrm{V} \end{bmatrix}_{\mathrm{Vp}}. \tag{11}$$

Figure 3.3: The grammar for Dutch, part I

```
% v -> []
```

$$\begin{bmatrix} \mathit{cat} : \mathrm{vp} \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{sc} : \mathrm{Sc} \\ \mathit{lex} : \mathrm{no} \\ \mathit{v2} : \begin{bmatrix} \mathit{cat} : \mathrm{vp} \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{sc} : \mathrm{Sc} \end{bmatrix} \\ \mathit{phon} : \mathrm{P} - \mathrm{P} \end{bmatrix}. \tag{6}$$

```
% comp -> omdat
```

$$\begin{bmatrix} \mathit{cat} : \mathrm{comp} \\ \mathit{sem} : \mathrm{omdat(Sem)} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{cat} : \mathrm{vp} \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{sc} : \langle\rangle \\ \mathit{v2} : \mathrm{no\_v2} \end{bmatrix} \rangle \\ \mathit{phon} : \text{“omdat”} \end{bmatrix}. \tag{8}$$

```
% adv -> vandaag
```

$$\begin{bmatrix} \mathit{cat} : \mathrm{adv} \\ \mathit{sem} : \mathrm{[vandaag](E)} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{sem} : \mathrm{E} \end{bmatrix} \rangle \\ \mathit{phon} : \text{“vandaag”} \end{bmatrix}. \tag{12}$$

```
% np -> arie
```

$$\begin{bmatrix} \mathit{cat} : \mathrm{np} \\ \mathit{sem} : \mathrm{arie} \\ \mathit{phon} : \text{“Arie”} \end{bmatrix}. \tag{14}$$

```
% np -> een dutje
```

$$\begin{bmatrix} \mathit{cat} : \mathrm{np} \\ \mathit{sem} : \mathrm{dutje} \\ \mathit{phon} : \text{“een dutje”} \end{bmatrix}. \tag{15}$$

Figure 3.4: The grammar for Dutch, part II

`% np -> leugens`

$$\begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{leugens} \\ \mathit{phon}: \text{“leugens”} \end{bmatrix}. \tag{16}$$

`% vp -> slaapt`

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{slaapt(Exp)} \\ \mathit{lex}: \text{yes} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sc}: \langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Exp} \end{bmatrix} \rangle \\ \mathit{phon}: \text{“slaapt”} \end{bmatrix}. \tag{2}$$

`% vp -> vertelt`

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{vertelt(Ag,Th)} \\ \mathit{lex}: \text{yes} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sc}: \langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Th} \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Ag} \end{bmatrix} \rangle \\ \mathit{phon}: \text{“vertelt”} \end{bmatrix}. \tag{3}$$

`% vp -> doet`

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{dutje\_doet(Exp)} \\ \mathit{lex}: \text{yes} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sc}: \langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Exp} \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \mathit{dutje} \end{bmatrix} \rangle \\ \mathit{phon}: \text{“doet”} \end{bmatrix}. \tag{13}$$

Figure 3.5: The grammar of Dutch, part III

refutation(Goal) :-
    rule(Goal, Ds),
    refutations(Ds).

refutations(⟨⟩).
refutations(⟨H|T⟩) :-
    refutation(H),
    refutations(T).

Figure 3.6: Meta interpreter for $\mathcal{R}(\mathcal{L})$-grammars

of Dymetman and Isabelle (1988), the order of generation is defined by the rule-writer by annotating DCG rules. In the approach of Wedekind (1988), this ordering is achieved automatically (and dynamically), essentially by a version of the goal-freezing technique (Colmerauer, 1982). Put simply, the generator only generates a node if its feature structure is instantiated; otherwise the generator will try to generate other nodes first.

A specific version of this approach, is an approach where one of the daughters has a privileged status. This daughter, that we might call the 'head' of the rule, will always be generated first. I will assume that for all rules the first daughter of the list of daughters represents the head. Note that this does not imply that this daughter is the leftmost daughter; the information about the left-to-right order of daughters is represented by the difference-list representation of strings in each node. Moreover note that this particular representation may be the result of some compilation step from a representation that is more convenient for the rule writer. For example, rule 1 was represented schematically as

$$sign(\mathrm{X}_0) \text{:-} sign(\mathrm{X}_1), sign(\mathrm{X}_2), \phi.$$

If we assume that the second daughter is the head, then this clause is simply written:

$$sign(\mathrm{X}_0) \text{:-} sign(\mathrm{X}_2), sign(\mathrm{X}_1), \phi.$$

Now without any modification to the original definition of *refutations* this change will imply that heads are generated first. Assume furthermore that the head of a rule is that daughter which shares its semantics with the mother node. In all (non-unit) rules of the Dutch example grammar it is possible to choose such a daughter.

The resulting simple top-down generation algorithm is equivalent to the approaches of Dymetman and Isabelle (1988) and Wedekind (1988) with respect to one major problem: the left-recursion problem. Consider what happens if we try to generate a sentence for the same semantic structure 17 as before, but this time assuming that heads are generated first. Again, the generator comes to the point where it tries to generate a saturated verb phrase with semantics slaapt(arie). As before the generator selects the binary verb phrase rule 1. This time the generator does not try to generate the argument of this rule, but immediately starts to generate its head. The resulting feature structure now looks as follows:

$$\begin{bmatrix} cat : \mathrm{vp} \\ sc : \langle \mathrm{Obj}_1 \rangle \\ sem : \mathrm{slaapt(arie)} \\ v2 : \mathrm{no\_v2} \end{bmatrix}$$

where Obj_1 is shared with the argument that still needs to be generated after the generation of the head. This feature structure can be the input for the verb phrase rule 1 again, of which the head will then be instantiated into:

$$\begin{bmatrix} cat : \mathrm{vp} \\ sc : \langle \mathrm{Obj}_2, \mathrm{Obj}_1 \rangle \\ sem : \mathrm{slaapt(arie)} \\ v2 : \mathrm{no\_v2} \end{bmatrix}$$

Each time the same rule can be applied (predicting longer and longer subcategorization lists), resulting in non-termination, because of left recursion. It should be clear that this non-termination problem is not necessarily caused by the leftmost daughter of a rule; the problem is caused by the node that is generated first. Thus the English equivalent of the verb phrase rule where the order of the head and the argument is switched presents exactly the same problem.

In this particular case, the problem arises because there is no limit to the size of the subcategorization list. Although one might propose an ad hoc upper bound on the length of the subcategorization list for lexical entries, even this expedient may be insufficient. In analyses of Dutch cross-serial verb constructions (Evers, 1975; Huybrechts, 1984) subcategorization lists such as these may be appended by syntactic rules (Moortgat, 1984; Steedman, 1985; Pollard, 1988; van Noord *et al.*, 1990) resulting in indefinitely long lists. Consider the Dutch sentence

(18) dat [Jan [Arie [Bob [de muizen [zag helpen loslaten]]]]
that Jan Arie Bob the mice saw help release
that Jan saw Arie help Bob release the mice

The string of verbs is analyzed by appending their subcategorization lists, as is illustrated in figure 3.7. Subcategorization lists under this analysis can have any length, and it is impossible to predict from a semantic structure the size of its corresponding subcategorization list merely by examining the lexicon.

In summary, top-down generation algorithms, even if controlled by the instantiation status of goals, can fail to terminate on certain grammars. The case given above, reminiscent of analyses from HPSG, as well as analyses from categorial unification grammar are examples in which the well-foundedness of the generation process resides in lexical information unavailable to top-down regimes. The conclusion of this section therefore is that top-down generators have problems with some linguistically motivated left recursive analyses. The source of the problem is that information available in lexical entries is not used sufficiently to guide the search.

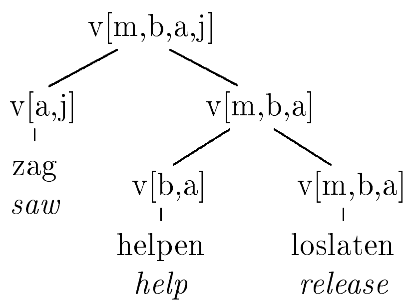


Figure 3.7: Analysis of cross-serial dependency, in which subcategorization lists of verbs are appended. The tree represents the verb-cluster of the sentence 'dat Jan Arie Bob de muizen zag helpen loslaten.

3.3.2 Shieber's chart-based generator

Shieber (1988) proposes a chart-based generator where rules are applied in a bottom up fashion. Results are kept on an Earley type chart (Earley, 1970). To make this process goal-driven, there is only one restriction: the logical form of every sub-phrase that is found, must subsume some part of the input logical form. This restriction results in the *semantic monotonicity* requirement on grammars. This restriction requires that the logical form of each daughter of a rule subsumes part of the logical form of the mother node of that rule. Note that this requirement also implements the *coherence* condition on the generation problem discussed in chapter 2: it is never possible that the generator comes up with an extension of the input semantic structure.

An example will clarify the strategy. Assume we want to generate a string for the logical form

omdat(vertelt(arie,leugens))

with the Dutch example grammar, where I leave the empty verb, to analyze *verb second* constructions, out of consideration for the moment. When the generator starts, it will try to select rules without any daughters (i.e. lexical entries), because the chart is still empty. The logical form of these entries should subsume some part of the input logical form.

First it can apply the rules *arie*, *leugens*, *vertelt* and *omdat*. Next, a verb phrase dominating *vertelt* and *leugens* will be constructed as well, with the semantic structure

vertelt(Subj,leugens)

A rule applies that combines the NP *arie* and the preceding verb phrase, resulting in the verb phrase *arie leugens vertelt*, with the logical form.

vertelt(arie,leugens)

This verb phrase can be the input for the complementizer rule, together with the complementizer *omdat*, resulting in the subordinate clause *omdat arie leugens vertelt*. This complementizer phrase has the appropriate logical form. Note that no other rules can apply, because their resulting logical form does not subsume part of the input logical form (remember we did not take into account the empty verb).

The requirement that every rule application yields a logical form that subsumes part of the input only results in a complete generation algorithm only if the grammar is semantically monotonic. Shieber admits that this requirement is too strong (op. cit. section 7):

> "Perhaps the most immediate problem raised by the methodology for generation introduced in this paper is the strong requirement of semantic monotonicity. (...) Finding a weaker constraint on grammars that still allows efficient processing is thus an important research objective."

In fact the grammar of the previous section is not semantically monotonic, because it analyses the Dutch idiomatic phrase 'een dutje doen' as a predicate without any internal structure, although in the derivation the logical form 'dutje' will be assigned to the noun phrase 'een dutje'.

Another example of an analysis that does not obey the semantically monotonicity requirement may be encountered with particle verbs. As an example consider the case where a sentence like

(19) jan berekent de kosten door
jan computes the costs through
jan computes the costs

has a logical form

doorberekenen(jan,kosten)

Other examples where semantic monotonicity is not obeyed are cases where semantically empty words such as 'there' and 'it' are syntactically necessary, and prepositional verbs such as 'count on', as in the following examples.

(20) a. There are mice in the hotel.

b. It has been raining for weeks.

c. It seems that Bob has ordered an ice-cream.

d. Arie always counts on Jan to count the costs.

Another disadvantage of Shieber's generator is the nondeterministic style of processing. The requirement, that only rules can be applied, of which the logical form subsumes some part of the input logical form, does not direct the generation process very much. Furthermore, the necessary subsumption checks (for example to check whether a result already is present in the chart) lead to much processing overhead.

Summarizing section 3.3, the principal problem of top-down generators is left-recursion. This problem is solved in a chart-based bottom-up generator at the cost of severe restrictions on possible grammars, and rather inefficient processing. These considerations led to the head driven bottom-up family of generators to be discussed in the next section.

3.4 Head driven bottom-up generation

In this section, I will present a simple variant of a head driven, bottom-up generator, called BUG, which was originally described in (van Noord, 1989), and shares many characteristics with the approaches presented in Calder *et al.* (1989), Shieber *et al.* (1989) and Shieber *et al.* (1990). These generation algorithms are characterized by a bottom-up style of processing, where top-down prediction is applied through the use of the notion '(semantic)-head'. Bottom-up processing can be motivated from the desire to use lexical information to guide the search, as much as possible. Head-driven processing can be motivated by the desire to use the semantics (the input) as an important source of information to guide the search.

I require that all non-unit rules have a head. Moreover, the logical form of this head must be *identical* with the logical form of the mother node; i.e. the mother node and the head *share* their logical form. Note, that for each non-unit rule in the Dutch example grammar, it is possible to choose a daughter as the head, that satisfies this requirement. As before, the head of the rule will be the first element of the list of daughters in the representation used by the meta-interpreter. Lexical entries are represented, for the meta-interpreter, as rules with an empty list of daughters.

The algorithm BUG proceeds as follows. Its input will be some node N that is associated with some semantic structure S. First BUG tries to find the *lexical head*, a lexical entry of which the semantics unifies with S. This part is called the *prediction* step. Now, BUG is going to build from this lexical head larger units as follows. It selects a rule of which the head unifies with the lexical head. The other daughters of this rule are generated recursively. For the mother node of this rule, this procedure will be repeated: selecting a rule of which the head unifies with the current node, generate the daughters of this rule and connect the mother node upward. This part of the algorithm ends if a mother node has been found that unifies with node N. This is defined in figure 3.8.

As an example, consider what happens if this algorithm is activated by the following query (again, I leave the empty verb out of consideration for the moment):

$$\texttt{?-}bug(\begin{bmatrix} cat : \text{comp} \\ sc : \langle\rangle \\ sem : \text{omdat(vertelt(arie,leugens))} \end{bmatrix}_{\text{C}}).$$

In figure 3.9 the flow of control of the generation process is illustrated. Firstly, the clause *predict_head* will select the lexical head, a lexical entry with a logical form that

$$
\begin{array}{l}
\textit{bug}(\mathrm{Goal})\,\texttt{:-}\\
\qquad \textit{predict\_head}(\mathrm{Goal}, \mathrm{Lex}),\\
\qquad \textit{sem\_head}(\mathrm{Lex}, \mathrm{Goal}).\\[1em]
\textit{sem\_head}(\mathrm{Goal}, \mathrm{Goal}).\\
\textit{sem\_head}(\mathrm{Head}, \mathrm{Goal})\,\texttt{:-}\\
\qquad \textit{select\_rule}(\mathrm{Head}, \mathrm{Mother}, \mathrm{Others}),\\
\qquad \textit{bug\_ds}(\mathrm{Others}),\\
\qquad \textit{sem\_head}(\mathrm{Mother}, \mathrm{Goal}).\\[1em]
\textit{bug\_ds}(\langle\rangle).\\
\textit{bug\_ds}(\langle \mathrm{H}|\mathrm{T}\rangle)\,\texttt{:-}\\
\qquad \textit{bug}(\mathrm{H}),\\
\qquad \textit{bug\_ds}(\mathrm{T}).\\[1em]
\textit{predict\_head}(\mathrm{Goal}, \mathrm{Lex})\,\texttt{:-}\\
\qquad \textit{head}(\mathrm{Goal}, \mathrm{Lex}),\\
\qquad \textit{rule}(\mathrm{Lex}, \langle\rangle).\\[1em]
\textit{head}(\left[\ \textit{sem} : \mathrm{Sem}\ \right], \left[\ \textit{sem} : \mathrm{Sem}\ \right]).\\[1em]
\textit{select\_rule}(\mathrm{Head}, \mathrm{Mother}, \mathrm{Ds})\,\texttt{:-}\\
\qquad \textit{rule}(\mathrm{Mother}, \langle \mathrm{Head}|\mathrm{Ds}\rangle).
\end{array}
$$

Figure 3.8: The $\mathcal{R}(\mathcal{L})$ definition of the simple version of BUG.

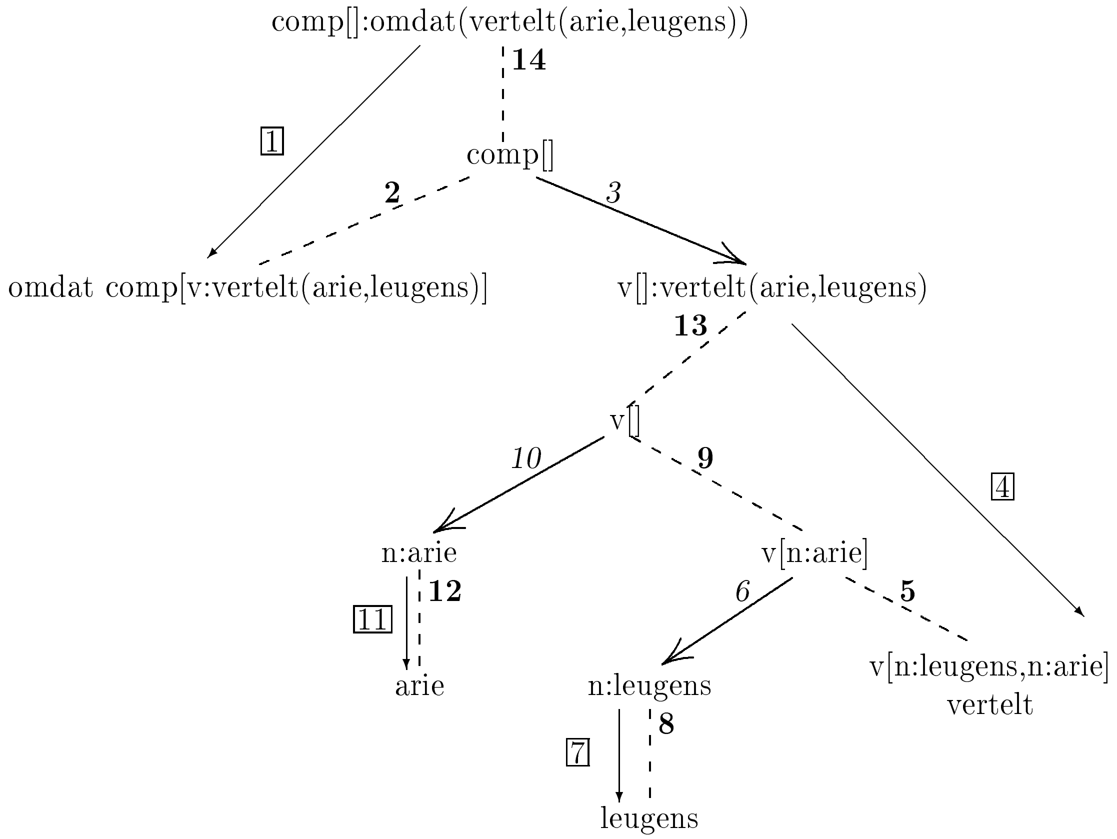


Figure 3.9: A trace of the generation of 'omdat Arie leugens vertelt'. The integers refer to the steps in the generation process. Framed integers on arrows with a small arrow head represent prediction steps, bold face integers on dashed lines represent connection steps, and slanted integers on arrows with a large arrow head represent recursive generation steps.

unifies with 'omdat(vertelt(arie,leugens))'. The definition of *omdat* (rule 8) is the only candidate (step 1). The prediction step instantiates this entry into:

$$\begin{bmatrix} \textit{cat} : \text{comp} \\ \textit{sem} : \text{omdat(vertelt(arie,leugens))} \\ \textit{sc} : \langle \begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \text{vertelt(arie,leugens)} \\ \textit{v2} : \text{no\_v2} \end{bmatrix} \rangle \\ \textit{phon} : \langle \text{omdat} | \text{P}_1 \rangle - \text{P}_1 \end{bmatrix}$$

It is important to note here, that not only the semantics of the lexical entry is instantiated, but also the semantics of the element on its subcategorization list. This entry needs to be connected upwards to the top-goal. To this end a rule is selected of which the head unifies with the feature structure of the complementizer. Rule 7 clearly is the only candidate (step 2). The next goal, therefore, is to generate the non-head daughter of this rule (3). Because of the subcategorization technique, this non-head daughter is unified with the element of the subcat list of the head. Therefore, the semantics of this non-head daughter is instantiated as well:

$$\begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \text{vertelt(arie,leugens)} \\ \textit{v2} : \text{no\_v2} \end{bmatrix}_{\text{T}}$$

Again, the algorithm predicts a lexical entry for this goal. The definition for *vertelt* (rule 3) is a possible candidate, obtaining the following feature structure (step 4):

$$\begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sem} : \text{vertelt(arie,leugens)} \\ \textit{lex} : \text{yes} \\ \textit{v2} : \text{no\_v2} \\ \textit{sc} : \langle \begin{bmatrix} \textit{cat} : \text{np} \\ \textit{sem} : \text{leugens} \end{bmatrix}, \begin{bmatrix} \textit{cat} : \text{np} \\ \textit{sem} : \text{arie} \end{bmatrix} \rangle \\ \textit{phon} : \langle \text{vertelt} | \text{P}_2 \rangle - \text{P}_2 \end{bmatrix}_{\text{S}}$$

Note again, that the logical form of the elements of the subcat list of this entry get instantiated as a result of the unification of the logical form of the goal, and the logical form of the verb.

The feature structure S is going to be connected to T by the *sem_head* clauses. To be able to connect the lexical VP to (ultimately) the saturated VP node, a rule will be selected of which this lexical verb phrase can be the head. Rule 1 is a possible candidate (5). If this rule is selected, the following feature structure represents the non-head daughter of the rule (again with instantiated semantics):

$$\begin{bmatrix} \textit{cat} : \text{np} \\ \textit{sem} : \text{leugens} \\ \textit{phon} : \text{P}_1 - \text{P}_2 \end{bmatrix}_{\text{D}}$$

The mother of the rule is instantiated as:

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{vertelt(arie,leugens)} \\ \mathit{lex}: \text{no} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sc}: \langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{arie} \end{bmatrix} \rangle \\ \mathit{phon}: \langle \text{vertelt} | \mathrm{P}_1 \rangle - \mathrm{P}_2 \end{bmatrix}_{\mathrm{M}}$$

The daughter D is generated recursively (6, 7, 8), instantiating $\mathrm{P}_1 - \mathrm{P}_2$ into $\langle \mathit{leugens} | \mathrm{P}_3 \rangle - \mathrm{P}_3$. Therefore the next task is to connect

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{lex}: \text{no} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sem}: \text{vertelt(arie,leugens)} \\ \mathit{sc}: \langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{arie} \end{bmatrix} \rangle \\ \mathit{phon}: \langle \text{leugens}, \text{vertelt} | \mathrm{P}_3 \rangle - \mathrm{P}_3 \end{bmatrix}_{\mathrm{M}}$$

upwards to the saturated verb phrase T. Again it is possible to choose rule 1 (step 9). In this case, the non-head daughter of the rule consists of the feature structure that results in the phonology 'arie' by a recursive application of BUG (10, 11, and 12); hence the mother node of this instantiation of rule 1 will become:

$$\begin{bmatrix} \mathit{cat}: \text{s} \\ \mathit{lex}: \text{no} \\ \mathit{v2}: \text{no\_v2} \\ \mathit{sem}: \text{vertelt(arie,leugens)} \\ \mathit{sc}: \langle \rangle \\ \mathit{phon}: \langle \text{arie}, \text{leugens}, \text{vertelt} | \mathrm{P}_3 \rangle - \mathrm{P}_3 \end{bmatrix}_{\mathrm{M}_2}$$

This node can easily be connected to the top node T by the first clause for *sem_head*, because it can be unified with the top node (step 13); thereby finishing the generation goal of the argument of the complementizer. The resulting complementizer phrase can also be connected trivially to the ultimate top goal C (14); the answer to the query, therefore, is:

$$\begin{bmatrix} \mathit{cat}: \text{comp} \\ \mathit{sem}: \text{omdat(vertelt(arie,leugens))} \\ \mathit{sc}: \langle \rangle \\ \mathit{phon}: \langle \text{omdat}, \text{arie}, \text{leugens}, \text{vertelt} | \mathrm{P}_6 \rangle - \mathrm{P}_6 \end{bmatrix}_{\mathrm{C}}$$

As another example, consider the case where the logical form is built in a semantically non-monotonic way:

$$\texttt{?-}\mathit{bug}(\begin{bmatrix} \mathit{cat}: \text{comp} \\ \mathit{sc}: \langle \rangle \\ \mathit{sem}: \text{omdat(dutje\_doet(bob))} \end{bmatrix}).$$

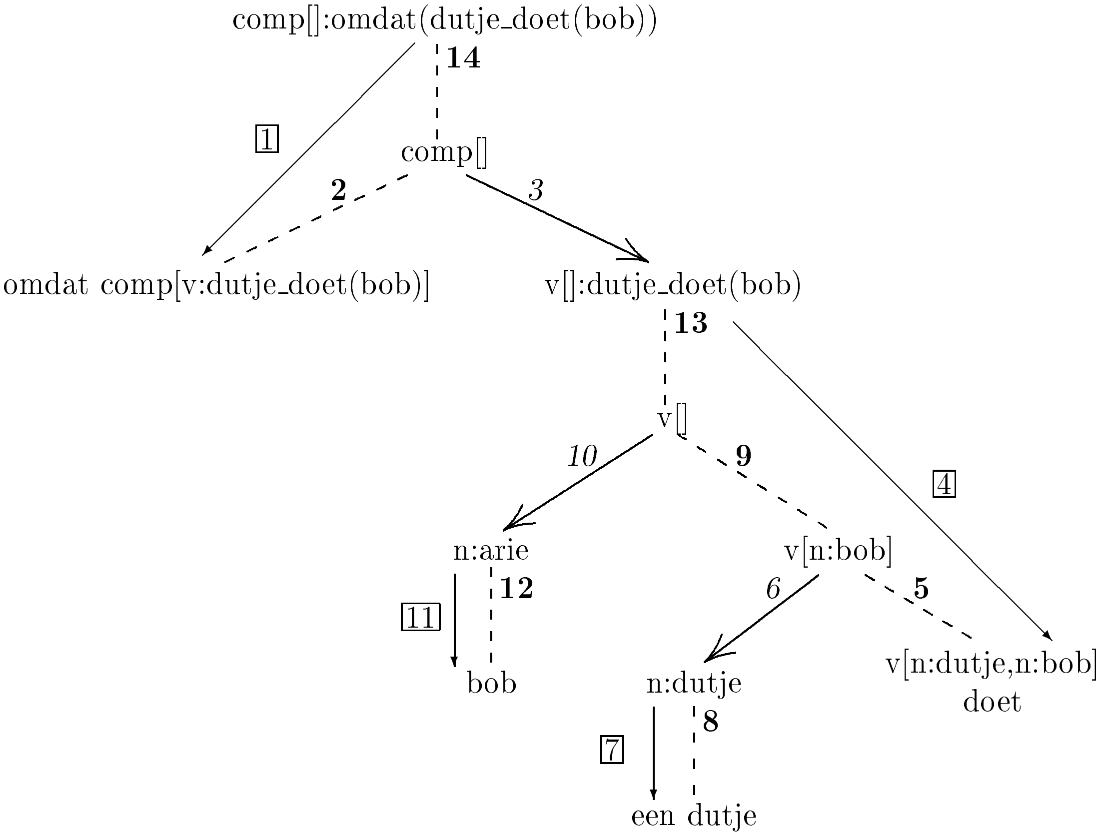


Figure 3.10: A trace of the generation of 'Omdat bob een dutje doet'; the integers are used as in the previous figure.

Again, the generation process can be illustrated as in figure 3.10. After the selection of the complementizer rule 7, the generator again tries to generate a saturated verb phrase recursively. For this generation goal, the predictor step selects the lexical entry *doet* (rule 13), after which the generator will try to connect the verb phrase:

$$\begin{bmatrix} \textit{cat} : \text{vp} \\ \textit{sem} : \text{dutje\_doet(bob)} \\ \textit{lex} : \text{yes} \\ \textit{v2} : \text{no\_v2} \\ \textit{sc} : \langle \begin{bmatrix} \textit{cat} : \text{np} \\ \textit{sem} : \text{bob} \end{bmatrix}, \begin{bmatrix} \textit{cat} : \text{np} \\ \textit{sem} : \text{dutje} \end{bmatrix} \rangle \\ \textit{phon} : \text{“doet”} \end{bmatrix}$$

to the saturated verb phrase goal, as in the foregoing example (step 5 in the illustration). Note, that the semantics of the two elements of the subcat list of 'doet' both are instantiated. Hence the algorithm proceeds exactly as in the preceding example, generating 'een dutje' and 'bob' both recursively. This results in the correct answer:

$$\begin{bmatrix} cat : \text{comp} \\ sem : \text{omdat(dutje\_doet(bob))} \\ sc : \langle\rangle \\ phon : \langle \text{omdat}, \text{bob}, \text{een}, \text{dutje}, \text{doet} | \text{P}_6 \rangle - \text{P}_6 \end{bmatrix}$$

Properties of the Algorithm

The algorithm BUG defines a simple bottom-up head-driven generation procedure. Note that the search is guided by the input because of the definition of the predict step. The definition of this step uses the knowledge that heads always share the logical form with their mother (remember this is how we defined the notion 'head'). Therefore, for any given top-goal it is possible and correct to share this information with the 'lexical head' immediately. The resulting system implements, therefore, an attractive compromise between bottom-up and top-down approaches; the order of processing is bottom-up, but there is an important information flow in top-down direction. Furthermore the recursive *bug* calls also provide for an important flow of information in top-down direction especially in the case of rules that dominate subcategorized-for elements as in the verb phrase rule 1. Later in this chapter I discuss extensions to BUG, for grammars in which other ways to construct semantic structures are assumed.

Note that the order of processing of the algorithm is not left-to-right, but *bi-directionally* because the algorithm always starts from the head of a rule. The logical form of this head is always known by the prediction step. This constitutes the top-down information of the algorithm. Apart from the top-down logical form information, the algorithm is directed by the information of the lexicon because the order of processing is bottom-up. Head driven bottom-up generators are thus geared towards the semantic content of the input on the one hand and lexical information on the other hand. Of course this is especially useful for grammars that are written in the spirit of *lexicalistic* linguistic theories such as CUG, UCG and HPSG.

Apart from considerations of efficiency the major reason for constructing *bottom-up* generators has been the left-recursion problem summarized in section 3.3. If the base case of the recursion resides in the lexicon the bottom-up approach does not face these problems. Typically in grammars that are based on the lexicalist theories mentioned above, these cases occur frequently, but are handled by BUG without any problems: the subcat lists are sufficiently instantiated to restrict the otherwise unlimited search.

Dymetman *et al.* (1990) define a grammatical formalism (called 'Lexical Grammars') in which the use of subcategorization lists and lexical construction of semantics as sketched above, is built in. This enables them to ensure, that generation terminates for grammars in which each of the lexical entries are defined in such a way, that the semantic structures of each of the elements on the subcat list is 'smaller' than the semantic structures of the lexical entry itself. The generation algorithm discussed here always terminates for this class of grammars.

Some possible extensions and some problems of BUG are discussed in the next sections.

3.5 Some Possible Extensions

Of course the simple architecture of BUG can be extended in several ways. In this section I discuss some possibilities. The next section will then be devoted to some problems BUG and its extensions face.

3.5.1 Restrictions on heads

The assumption, that heads always share their logical form with the mother node, may be too restrictive for specific linguistic or semantic theories. Some extensions to BUG are possible that handle more sophisticated grammars. For example it is possible, as proposed in van Noord (1989), to enlarge the power of the prediction step. By inspection of the grammar it may be possible to pre-compile possible relations between the logical form of some top node and the logical form of the lexical head of that node. In that case, the notion 'semantic-head' is defined in some other way than by the requirement that its logical form is shared with the mother node. In the general case a pre-compiler then needs to compute the reflexive and transitive closure of the relation between mothers and heads, which may not always be without problems.

Another extension is the architecture advocated in Shieber *et al.* (1989); Shieber *et al.* (1990), where rules are divided in two types. The first type of rules, are rules where some daughter indeed shares its logical form with the mother node. Such rules are called *chain-rules*, and the relevant daughter is called the 'head', as before. In the second type of rule there is no such daughter. These rules are called 'non-chain-rules'. By this definition, all lexical entries are non-chain-rules because they have no daughters, and hence no head daughter. The algorithm does not necessarily predict a lexical entry, but it predicts a non-chain-rule. The daughters of this rule are then generated in a top-down fashion. After the generation of these daughters, the mother node of the non-chain-rule is connected to the top node bottom-up, as in BUG. In case all non-unit rules of a grammar are chain-rules, the algorithm is equivalent to BUG. In case no rules have a head, the algorithm reduces to a top-down generator. Assume temporarily that clauses are represented as follows:

$ncr(\text{Mother}, \text{Ds})\,\texttt{:-}\,\phi.$

for non-chain-rules where Mother is the mother node, and Ds is a list of daughter nodes (which will be empty for lexical entries). Chain-rules are represented as:

$cr(\text{Head}, \text{Mother}, \text{Ds})\,\texttt{:-}\,\phi.$

where Head is the head daughter of the rule, Ds is the list of non-head daughters. The extended algorithm is defined by replacing the definitions of *predict_head* and *select_rule*, of program 3.8, with the definitions of 21.

(21) $\mathit{predict\_head}(\mathrm{Goal}, \mathrm{IntermGoal})$:-
$\mathit{head}(\mathrm{Goal}, \mathrm{IntermGoal})$,
$\mathit{ncr}(\mathrm{IntermGoal}, \mathrm{Ds})$,
$\mathit{bug\_ds}(\mathrm{Ds})$.

$\mathit{select\_rule}(\mathrm{Head}, \mathrm{Mother}, \mathrm{Ds})$:-
$\mathit{cr}(\mathrm{Head}, \mathrm{Mother}, \mathrm{Ds})$.

3.5.2 Extending the prediction step

The prediction step, as it is defined in BUG, only uses *semantic* information. However, it is possible to extend the prediction step to take into account *syntactic* information as well. This is especially useful for grammars that define words that are semantically empty. For example, assume our Dutch grammar is extended with the complementizer *dat* (that) as defined in 22.

$$(22)\quad \mathit{sign}(\begin{bmatrix} \mathit{cat} : \mathrm{comp} \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{cat} : \mathrm{vp} \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{sc} : \langle\rangle \\ \mathit{v2} : \mathrm{no\_v2} \end{bmatrix} \rangle \\ \mathit{phon} : \text{“dat”} \end{bmatrix}).$$

In this entry, the semantics of the embedded verb phrase is simply ‘taken over’ as the semantics of the complementizer, and hence of the complementizer phrase headed by ‘dat’. Such lexical entries will be candidates for each invocation of the prediction step, because they are completely ignorant as to what semantics they may end up with. This leads to gross efficiency problems as has been observed in practice.

Suppose, however, that the prediction step is augmented with syntactic information. If the goal is to generate a verb phrase, then it is obvious from the grammar, that it is useless to predict a complementizer at that point, because the only results of connecting a complementizer upwards will be a complementizer phrase. Assume that the predicate $\mathit{link}(Moth, \mathrm{Head})$ is a pre-compiled table of the reflexive and transitive closure of possible syntactic links between mothers and heads, similar to the link predicate in the BUP parser (Matsumoto *et al.*, 1983) between mothers and left-most daughters. For the example grammar of this chapter, the predicate can be defined as follows, if we restrict the link table, to take into account only the value of the attribute *cat*:

$$(23)\quad \mathit{link}(\begin{bmatrix} \mathit{cat} : \mathrm{root} \end{bmatrix}, \begin{bmatrix} \mathit{cat} : \mathrm{q} \end{bmatrix}).$$
$$\mathit{link}(\begin{bmatrix} \mathit{cat} : \mathrm{q} \end{bmatrix}, \begin{bmatrix} \mathit{cat} : \mathrm{vp} \end{bmatrix}).$$
$$\mathit{link}(\begin{bmatrix} \mathit{cat} : \mathrm{root} \end{bmatrix}, \begin{bmatrix} \mathit{cat} : \mathrm{vp} \end{bmatrix}).$$
$$\mathit{link}(\begin{bmatrix} \mathit{cat} : \mathrm{X} \end{bmatrix}, \begin{bmatrix} \mathit{cat} : \mathrm{X} \end{bmatrix}).$$

It is possible to change the definition of *predict_head* and *select_rule* into the definitions given in 24.

(24) *predict_head*(Goal, IntermGoal) :-
head(Goal, IntermGoal),
link(Goal, IntermGoal),
ncr(IntermGoal, Ds),
bug_ds(Ds).

select_rule(Head, Mother, Ds, Goal) :-
link(Goal, Mother).
cr(Head, Mother, Ds).

Note that we change the *select_rule* clause to take four arguments now. The fourth argument is the top goal (i.e. the second argument of the *sem_head* clauses).

In fact, the prediction step does not necessarily have to be restricted to semantic and syntactic information, as long as it is possible to pre-compile the reflexive and transitive closure of the relation between heads and mothers. The 'restrictor' technique discussed by Shieber (1985) can be used here. Note though that the semantic information should not be 'restricted'. Syntactic prediction limits the choice of possible lexical entries. The semantic prediction has a further task in *instantiating* the semantics of the lexical entry to ensure that recursive generation calls also have their semantics specified. This difference is the reason to differentiate in the foregoing definition of *predict_head* between the semantic prediction and the syntactic prediction.

The syntactic linking technique has a problem in that it may produce spurious ambiguities. For example, consider the following part of a hypothetical syntactic linking table:

$$\vdots$$

$$\mathit{link}(\left[\ \mathit{cat}: \begin{bmatrix} \mathit{maj}: \mathrm{s} \\ \mathit{main}: \mathrm{yes} \end{bmatrix}\ \right], \left[\ \mathit{cat}: \left[\ \mathit{maj}: \mathrm{np}\ \right]\ \right]).$$

$$\mathit{link}(\left[\ \mathit{cat}: \begin{bmatrix} \mathit{maj}: \mathrm{s} \\ \mathit{mood}: \mathrm{decl} \end{bmatrix}\ \right], \left[\ \mathit{cat}: \left[\ \mathit{maj}: \mathrm{np}\ \right]\ \right]).$$

$$\vdots$$

Assume a goal is specified for category s, and the current category, that needs to be connected to that goal, is np. In that case both clauses are applicable. However, these two clauses do not necessarily correspond to two distinct results. In a Prolog implementation this problem may be solved by a 'double negation' trick; i.e. we would write something like:

```
\+ \+ link(Goal,Mother),
```

where `\+` is the negation-as-failure operator. The linking predicate then reduces to a check that does not introduce new information, but only filters the application of rules and lexical entries that cannot be linked to the goal (for that reason the predicate should be called immediately *after* the predicates *ncr* and *cr*, rather than before.). Doubly negating the semantic prediction step would, of course, be damaging, because the main task of the semantic prediction is to further instantiate (the semantics of) the predicted lexical entries.

Another possibility, to prevent spurious ambiguities of the syntactic linking device, would be to compute the generalization of all possible answers every time the 'link' predicate is called. In the previous example this would simply result in neglecting the *main* and *mood* attributes. In general such a solution requires quite a bit of overhead. For different grammars different answers will be possible to the question what the most efficient solution is.

3.5.3 Memo relations

The head-driven generator, defined here as a $\mathcal{R}(\mathcal{L})$ meta-interpreter, uses the depth-first backtrack search strategy described in section 2.3. The motivation of the head-driven generation strategy lies in the *reduction* of the search space that is obtained. How to search this reduced search space is quite an independent matter. Therefore, it may be useful to investigate alternative search strategies. For example, the algorithm could be defined using a chart, as for example proposed in Calder *et al.* (1989); Gerdemann (1991).

Another, related, possibility consists of the implementation of so-called 'memo-relations' (also called 'well-formed sub-string tables' in the context of parsing). The idea is that, to prove some goal Goal, we first check whether such a goal has already been proven before. In that case we pick up the results of that previous call. If the goal has not been tried before, we will go about finding all solutions to this goal (and assert each of them in the database). If no more solutions can be found, then we assert that all solutions for that goal have been found, and we pick up a possible solution. This in effect turns the search strategy into a breadth-first one.

In figure 3.11 a possible implementation of memo-relations is defined in Prolog. This implementation improves upon the one discussed in Matsumoto *et al.* (1983); Pereira and Shieber (1987) in that the implementation is much more general (not restricted to parsing), and that results are indexed with respect to the goal which produces that result. This implies that subsumption checks are limited to goals, and no subsumption checks on full results is necessary. Clearly, goals are (much) 'smaller' than results in practice, and hence the costs of overhead is reduced. It should be stressed, though, that in the case of constraint-based grammars it is clear that such memo-relations do not improve on the worst-case behavior of the program (as there generally is an unbounded number of possible categories), and maybe not even on the practical behavior of the program (because of the overhead involved in copying and subsumption checking).

Even though the overhead involved in the implementation of such memo-relations is still considerable, it turns out that for many grammars the head-driven generator is more efficient if implemented as a memo-relation, along the lines of figure 3.11. For

```
memo(Goal):-
    copy_term(Goal,Frozen),    % make copy of goal
    numbervars(Frozen,0,_),    % and numbervar it
    memo(Goal,Frozen).         % for subsumption checks later

memo(Goal,Frozen) :-
    done(Frozen),              % more general goal is known
    !,
    item(Frozen,_,Goal).       % get result w.r.t. goal
memo(Goal,Frozen) :-
    copy_term(Goal,Result),    % copy such that goal does not change
    retractall(item(_,Goal,_), % retract more specific goals
    call(Result),              % find a result
    assertz(item(Goal,Frozen,Result)), % assert it
    fail.                      % fail to find other results
memo(Goal,Frozen) :-
    assertz(done(Goal)),       % assert that goal is known now
    item(Frozen,_,Goal).       % get result w.r.t. goal
```

Figure 3.11: The predicate `memo(Goal)` searches for `Goal` using a tabular breadth-first search technique. Results are asserted as `item(Goal,FrozenGoal,Result)` where `FrozenGoal` is a numbervarred copy of `Goal` for later subsumption checks. Goals are asserted as `done(Goal)`. Such a goal is only asserted once all solutions are found. Hence, the resulting program has the same termination properties. As a further improvement of this technique a user-provided definition of what constitutes the real 'goal' information of a given goal might be investigated. This would e.g. provide a hook for Shieber's restriction technique. Furthermore, this would allow for the incorporation of more complex constraints for which subsumption checking would not be possible.

example this technique has been used in order to improve upon the efficiency of the head-driven generation algorithm by a factor 4 for typical grammars.

3.5.4 Delay of lexical choice

The generation algorithm defined above chooses particular lexical forms on-line. This approach can lead to a certain amount of unnecessary nondeterminism. The choice of a particular form depends on the available semantic and syntactic information. Sometimes there is not enough information available, to choose a form deterministically. For instance, the choice of verb form might depend on syntactic features of the verb's subject, available only after the subject has been generated. This nondeterminism can be eliminated by deferring lexical choice to a post-process. Inflectional and orthographical rules are only applied when the generation process is finished and all syntactic features are known. In short, the generator will yield a list of lexical items instead of a list of words. To this list the inflectional and orthographical rules are applied.

The MiMo2 system (van Noord *et al.*, 1990; van Noord *et al.*, 1991) incorporates such a mechanism into the previous generation algorithm quite successfully. Experiments with particular grammars of Dutch, Spanish, and English have shown that the delay mechanism results in a generator that is faster by a factor of two or three on short sentences. Of course, the same mechanism could be added to any of the other generation techniques discussed in this chapter; it is independent of the traversal order.

The particular approach to delaying lexical choice found in the MiMo2 system relies on the structure of the system's morphological component as presented in figure 3.12.

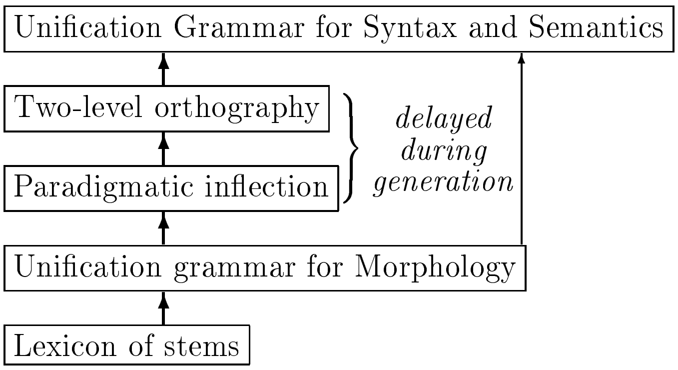


Figure 3.12: The architecture of MiMo2, in which lexical choice is delayed during generation.

The figure shows how inflectional rules, orthographical rules, morphology and syntax are related: orthographical rules are applied to the results of inflectional rules. These inflectional rules are applied to the results of the morphological rules. The result of the orthographical part are then input for the syntax. However, in the lexical-delayed scheme, the inflectional and orthographical rules are delayed. During the generation process the results of the morphological grammar are used directly. It should be emphasized that this is possible, only because the inflectional and orthographical rules are monotonic, in the sense that they only *further instantiate* the feature structure of a lexical item, but do not change it. This implies, for example, that a rule that relates an active and a passive variant of a verb will not be an inflectional rule but rather a rule in the morphological grammar (as it changes for example the subcategorization requirements of the verb). The rule that builds a participle from a stem may in fact be an inflectional rule if it only instantiates the feature *vform*, for instance. When the generation process proper is finished the delayed rules are applied and the correct forms can be chosen deterministically.

The delay mechanism is useful in the two following general cases:

Firstly, and most importantly, the mechanism is useful if an inflectional variant depends on syntactic features not yet available. The particular choice of whether a verb has singular or plural inflection, depends on the syntactic agreement features of its subject; these may only be available after the subject has been generated. Other examples may include the particular choice of personal and relative pronouns, and so forth.

Secondly, delaying lexical choice is useful when there are several variants for some word, that are equally possible, because they are semantically and syntactically identical. For example, a word may have several spelling variants. If we delay orthography then the generation process computes with only one "abstract" variant. After the generation process is completed, several variants can be filled in for this abstract one. Examples from English include words that take both regular and irregular tense forms (e.g. "burned/burnt"); and variants such as "traveller/traveler," "realize/realise," etc.

3.5.5 Other improvements

The generation algorithm can be improved somewhat further by the application of standard logic programming techniques. In Block (1991) several optimizations for semantic-head-driven generation are described. For example, non-chain-rules with variable semantics (such rules may correspond to traces, or function words), are eliminated by a partial evaluation technique. Furthermore, a better indexing on semantic structures is described in order to have fast access to non-chain-rules, whereas chain-rules are indexed on their syntactic features.

Another possible improvement is discussed in Russell *et al.* (1990); they describe a variant of the semantic-head-driven generation algorithm in which the semantic head of a phrase is connected upward, but the generation of the daughters of each chain-rule is only performed once the head has been connected upward. This effect can be obtained by re-ordering the clauses of the *sem_head* predicate as follows:

(25) *sem_head*(Goal, Goal).
    sem_head(Head, Goal) :-
        select_rule(Head, Mother, Others),
        sem_head(Mother, Goal),

bug_ds(Others).

Finally, note that defining the head-driven generator as a meta-interpreter may at first sight seem somewhat inefficient. However note that such a meta-interpreter can often be compiled away. In Ruessink and van Noord (1989) such a compilation is described for a semantic-head-driven generator defined as a meta-interpreter in Prolog. The resulting program is very similar to the result of grammar compilation described in Dymetman *et al.* (1990); Dymetman (1991).

3.6 Problems for BUG

3.6.1 Verb-second

Although I did not give any examples BUG can correctly handle analyses that make use of empty elements, such as in the case of a *gap threading* analysis of *wh-movement*, *topicalization* and *relativization*. However, there is one general exception to this claim. In case the head itself has been 'moved', there may be a problem for BUG. Consider the analysis of *verb second* phenomena in Dutch and German. As we discussed shortly in the beginning of this chapter, it is assumed in most traditional analyses that the verb in root sentences has been 'moved' from the final position to the second position. Koster (1975) convincingly argues for this analysis of Dutch. Thus a simple root sentence in German and Dutch usually is analyzed as in the following Dutch examples:

(26) a. Vandaag vertelt$_i$ Arie Bob leugens ϵ_i
Today tells Arie Bob lies
Today Arie tells lies to Bob

b. Vandaag heeft$_i$ Arie Bob leugens ϵ_i verteld
Today has Arie Bob lies told
Today Arie has told lies to Bob

c. Vandaag [hoort en begrijpt]$_i$ Bob Arie's leugens ϵ_i
Today hears and understands Bob Arie's lies
Today Bob hears and understands Arie's lies

As explained in section 3.2 this can easily be implemented in a unification-based grammar, using a simple threading technique and an empty verb. Rule 5 consists of a verb and a verb phrase that misses this verb. There is some freedom in choosing the head of this rule. If it is the case that the verb always is the semantic head of rule 5 then BUG can be made to work properly if the prediction step includes information about the verb second position that is percolated via the other rules (the *v2* attribute). In general however, the verb will not be the semantic head of the sentence, as is the case in this grammar. Because of rule 11, the verb can have a different logical form compared to the logical form of the mother of rule 5. This leads to a problem for BUG. The problem comes about because BUG can (and must) at some point predict the empty verb as the lexical head of the construction. However in the definition of this empty verb no information (such as the list of complements) will get instantiated (unlike in the usual case of lexical entries). Therefore rule 1 can be applied an unbounded number of times. The length of the lists of complements now is not known in advance, and BUG will not terminate.

In van Noord (1989) an ad-hoc solution is proposed. This solution assumes that the empty verb is an inflectional variant of a verb. Moreover inflection rules are applied when the generation process proper, is finished (and has yielded a list of lexical entries), as described in the previous section. During the generation process the generator acts as if the empty verb is an ordinary verb, thereby circumventing the problem. However this solution only works if the head that is displaced is always a lexical entry. This is not true in general. In Dutch the verb second position can not only be filled by (lexical) verbs but also by a conjunction of verbs (cf. 26c). Moreover it seems to be the case that Spanish is best analyzed by assuming the 'movement' of complex verbal constructions to the second position (*verb phrase second*).

A more general solution is proposed in van Noord (1990a); Shieber *et al.* (1990). In this solution it is assumed that there is a relation between the empty head of a construction, and some other construction (in the case of verb second, the verb in second position). However the relation is usually implicit in a grammar; it comes about by percolating the information through different rules from the verb second position to the verb final position. In the proposal under discussion this relation is made explicit by defining an empty head as a clause with two arguments as in 27.

$$\text{(27)}\quad \textit{head\_gap}(\begin{bmatrix} \textit{cat}: \text{vp} \\ \textit{lex}: \text{yes} \\ \textit{v2}: \text{no\_v2} \\ \textit{sem}: \text{Sem} \\ \textit{sc}: \text{Sc} \end{bmatrix}_{\text{Ant}}, \begin{bmatrix} \textit{cat}: \text{vp} \\ \textit{lex}: \text{no} \\ \textit{v2}: \begin{bmatrix} \textit{cat}: \text{vp} \\ \textit{sem}: \text{Sem} \\ \textit{sc}: \text{Sc} \end{bmatrix} \\ \textit{sem}: \text{Sem} \\ \textit{sc}: \text{Sc} \\ \textit{phon}: \text{P} - \text{P} \end{bmatrix}_{\text{Gap}}).$$

This definition can intuitively be understood as follows: once you have found some node Ant (the first argument of *head_gap*), then there could have been just as well the (gap-) node Gap (the second argument of *head_gap*). Note that a lot of information is shared between the two nodes, thereby making the relation between the antecedent and the empty verb explicit. The use of such rules can be incorporated in BUG by adding the following clause for *sem_head*:

(28) *sem_head*(Head, Goal) :-
head_gap(Head, Gap),
sem_head(Gap, Goal).

Note that the problem is now solved because the rule for the gap will only be selected, after its antecedent has been built. Some parts of this antecedent are then unified with some parts of the gap. The subcat list, for example, will thus be instantiated in time. The sentence

(29) Vandaag vertelt Arie leugens
Today tells Arie lies
Today Arie tells lies

is generated using this technique as in figure 3.13.

On the other hand, this solution can be criticised because it requires some information to be stated redundantly. Furthermore, if several different head_gaps are defined then the generator may yield spurious ambiguities. In the next chapter a completely different analysis of verb second is proposed that is not problematic for a head-driven generator of the type discussed here. In this analysis grammar rules may combine their daughters by more complex string operations than those usually allowed in unification grammars. In the current grammar, strings are combined by (a difference-list implementation of) concatenation. In the next chapter more powerful string combination operations are investigated.

3.6.2 Raising-to-object

Although I have argued that the *heads first* approach usually implies, that the logical form of a node is sufficiently instantiated at the time this node has to be generated, it is possible to write linguistically motivated grammars, where this is not the case. For example, *raising-to-object* constructions in English can be analyzed in a way, that is problematic for BUG. Assume that the semantic structure for the sentence

(30) Arie believes Bob to order a salad with ice

will be something like

believe(arie,order(bob,[with(ice)](salad)))

Furthermore assume that *Bob* is the syntactic object of *believes*. A reasonable definition (inspired by the LFG treatment of such cases) of the lexical entry *believes* in the spirit of the preceding Dutch example grammar then is the following:

$$\begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{believes}(\text{Arg}_1,\ \text{Arg}_2) \\ \mathit{sc}: \left\langle \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Obj} \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \text{vp} \\ \mathit{sem}: \text{Arg}_2 \\ \mathit{sc}: \left\langle \begin{bmatrix} \mathit{cat}: \mathit{np} \\ \mathit{sem}: \text{Obj} \end{bmatrix} \right\rangle \end{bmatrix}, \begin{bmatrix} \mathit{cat}: \text{np} \\ \mathit{sem}: \text{Arg}_1 \end{bmatrix} \right\rangle \\ \mathit{phon}: \text{“believes”} \end{bmatrix}$$

This lexical entry has, apart from its subject, two complements, a noun phrase and a verb phrase. Furthermore it is stated that the logical form of the noun phrase is identical to the logical form of the subject of the embedded verb phrase.

However, the previous analysis of raising-to-object constructions is problematic for BUG. Because the generator proceeds bottom-up it will try to generate the object

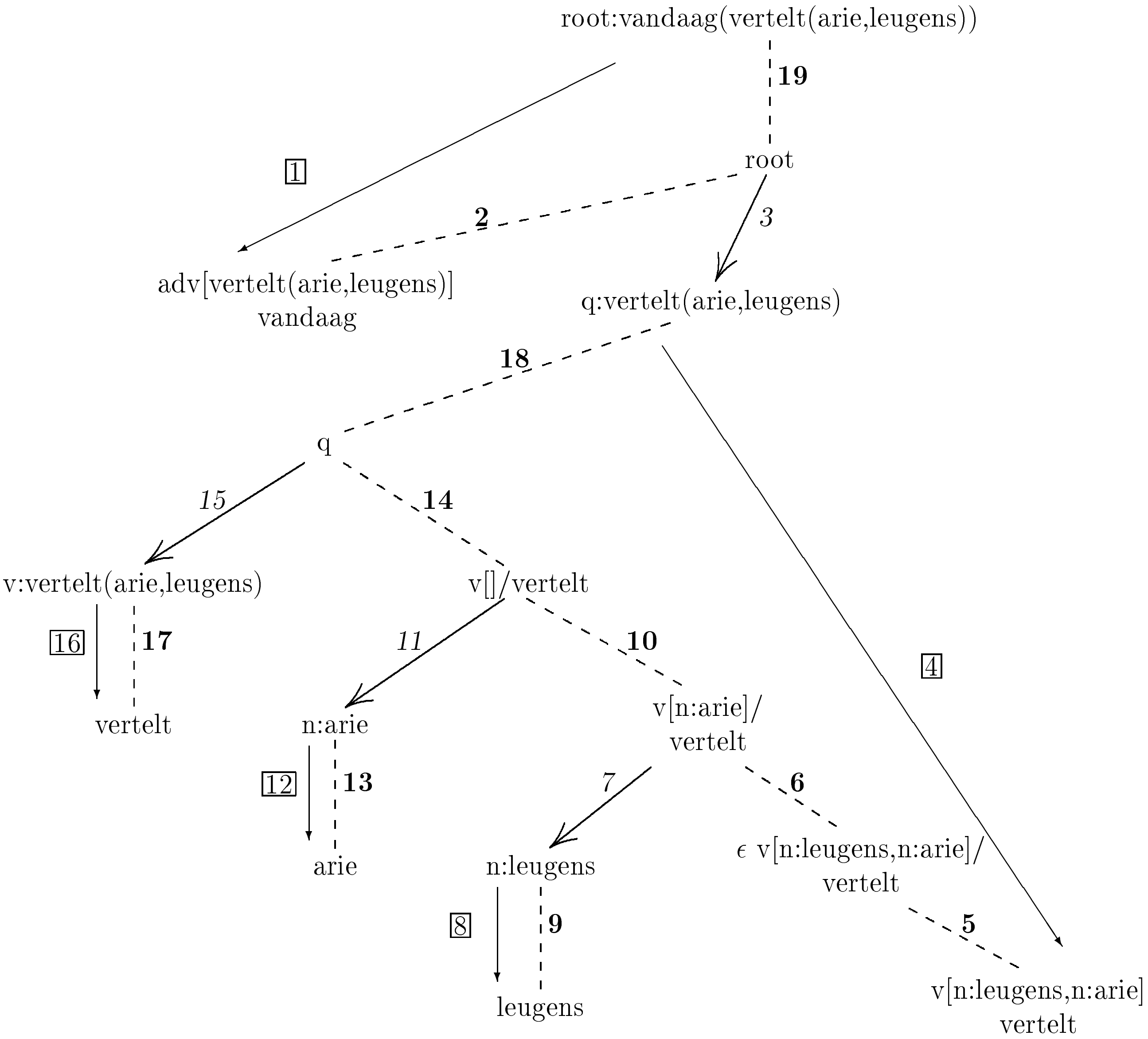


Figure 3.13: Trace of the generation of the verb-initial clause 'vandaag vertelt arie leugens'. In step number 4, the ordinary finite verb is predicted as the head of the verb phrase. In step number 5, the generator uses the definition of 'head_gap' to 'replace' this verb with an empty element, in which the information of the verb is instantiated. Finally, in step number 15, the verb 'vertelt' is generated as a non-head daughter.

noun phrase, *before* the embedded verb phrase has been generated, i.e., before the link between the embedded subject and the object is found. As a result, the logical form of the object is not yet instantiated, and therefore BUG may not terminate in this case, or at least waste much effort searching for all kinds of NP's that later turn out to be not the right ones. Assuming that the analysis of raising-to-object is correct, then it might be necessary to augment BUG with some version of *goal freezing*. If the generator comes across an un-instantiated logical form, then the execution of that node is suspended until the logical form is instantiated. In the case of *believes*, this will imply that the embedded verb phrase will be generated first, after which the object can be generated.

On the other hand, it has also been argued that the order in which the two objects of *believe* are selected is exactly the other way around as in the definition above. In Bach (1979) it is argued, that the verb phrase is selected *before* the NP object. Such an analysis can be motivated by observing that certain constraints on control, reflexivization and passive, are easily explained using the notion 'obliqueness', which in our example grammar is implemented as an ordering on the elements of the subcat list (see also Hepple (1990)). This analysis is only available in formalisms that provide for *wrapping* operations. Note that, if indeed the noun phrase object is selected *after* the verb phrase, no problem for BUG arises. In the next chapter we propose the use of more powerful string operations, such as wrapping. In such an extended formalism it is thus possible to analyze raising-to-object constructions such that it is not necessary to extend BUG with a dynamic computation rule as discussed above.

3.7 Conclusion

In this chapter I have presented a simple bottom-up and head-driven generation algorithm, and I discussed some extensions to it. I have argued that head driven bottom-up generators are useful for two reasons. Firstly, the order of processing is geared towards the semantic content of the input, and the information in the lexicon. Secondly, this order of processing puts less restrictions on grammars then top-down generators and Shieber's chart-based bottom-up generator. For grammars written in the spirit of lexical, sign-based linguistic theories head driven bottom-up generators seem especially useful.

The two sources of directedness (semantic information, and lexical information) yield generators with acceptable performance. For example, a version of the generation algorithm has been used successfully in the MiMo2 translation prototype (cf. chapter 5), for Dutch, Spanish and English. Furthermore, the same approach to generation has been used successfully in the CLE system (Alshawi and Pulman, 1992).

Certain analyses raise problems, as I discussed, for the proposed generation regime. In case heads are displaced a head-driven generator may face problems (as we saw in the analysis of verb-second in Dutch). Another problem occurs if the semantics of a phrase is dependent on a phrase which is generated *after* it, as we saw in the case of an LFG type analysis of raising-to-object. Specific extensions of the head-driven generator can be proposed to handle these problems. A more general solution can be offered in grammars in which operations on strings are allowed that go beyond concatenation. In such grammars analyses of both head-movement and raising-to-object are available which are un-problematic for head-driven generators of the type discussed here. The next chapter argues for such more powerful operations on strings, and provides a head-driven parsing algorithm for grammars employing such operations.

Chapter 4

Head-corner Parsing for Discontinuous Constituency

I describe a head-driven parser for a class of grammars, that handle discontinuous constituency by a richer notion of string combination than ordinary concatenation. The parser is a generalization of the left-corner parser and can be used for grammars written in powerful formalisms such as non-concatenative versions of UCG and HPSG, and lexicalized and constraint-based versions of Tree Adjoining Grammars.

4.1 Introduction

4.1.1 Discontinuous Constituency and Reversibility

Although most constraint-based formalisms in computational linguistics assume that phrases are built by concatenation (eg. as in PATR II, GPSG, LFG and most versions of Categorial Grammar) this assumption is sometimes challenged by allowing more powerful operations to construct strings. The linguistic motivation for such alternative conceptions of string combination are the analyses of so-called discontinuous constituency constructions. For example, Pollard (1984) proposes several versions of 'head wrapping'. In the analysis of the Australian free word-order language Guugu Yimidhirr, Mark Johnson uses a 'combine' predicate in a DCG-like grammar that corresponds to the union of words (Johnson, 1985). Mike Reape uses an operation called 'sequence union' to analyze Germanic semi-free word order constructions (Reape, 1989; Reape, 1990). Other examples include Tree Adjoining Grammars (Joshi *et al.*, 1975; Vijay-Shanker and Joshi, 1988), and versions of Categorial Grammar (Bach, 1979; Zwicky, 1986; Dowty, 1990; Hoeksema, 1991). Apart from the motivation from the syntax of discontinuous constituency, non-concatenative grammatical formalisms may also be motivated from a semantic perspective, as it is expected that such formalisms facilitate a systematic, compositional construction of semantic structures.

The use of non-concatenative grammars is furthermore motivated by the desire to obtain reversible grammars. This motivation is essentially twofold.

Motivation from generation. It is expected that the extra power available in non-concatenative formalisms, facilitates a systematic, compositional construction of semantic representations. Therefore, it will be easier to define generation algorithms. The semantic-head-driven generation strategy discussed in the previous chapter faces problems in case semantic heads are 'displaced', and this displacement is analyzed using threading. However, in this chapter I sketch a simple analysis of verb-second (an example of a displacement of semantic heads) by an operation similar to head wrapping which a head-driven generator processes without any problems (or extensions) at all.

Motivation from parsing. It is expected that non-concatenative grammars are useful for parsing as well. The parsing problem for grammars, written in concatenative formalisms such as PATR and DCG, is undecidable in general. Thus, the restriction that phrases are built by concatenation is not a 'real' restriction from a formal point of view. Often, it is possible to see whether such a grammar in fact can be parsed effectively. The 'dangerous' parts of a grammar are rules with an empty right-hand-side, and non-branching rules. Inspection of the grammar, and most notably its dangerous parts, sometimes may reveal that no problems arise. To analyze discontinuous constituency, the grammar writer is forced to use complicated 'gap threading' mechanisms (Pereira, 1981). Gap threading, though, heavily uses the 'dangerous' types of rule. For this reason, the more discontinuous constituency constructions are analyzed, the more difficult it becomes to see whether the resulting grammar can be used effectively for parsing. Furthermore, if at a certain moment the addition of a certain threading mechanism (say, for extraposition) *does* result in a grammar that is not effectively parsable anymore, it is unclear whether to blame the proposed extension to the grammar, or whether one of the other threading mechanisms should be blamed, (or whether the problem simply comes about because of the interaction of different threading mechanisms).

For this reason, non-concatenative grammars are motivated, because these grammars allow for more expressive power. This addition of expressive power may furthermore reduce the need of 'dangerous' rules, and thus non-concatenative grammars are useful for extendability.

4.1.2 Overview

This chapter is organized as follows. Firstly I discuss the proposals for more powerful string operations, as presented by Pollard (1984), Johnson (1985), Reape (1990), Vijay-Shanker and Joshi (1988) and Abeille (1988). Then I define two restrictions on possible string combinations for constraint-based grammars, based on Vijay-Shanker *et al.* (1987). The combination of strings is restricted to be linear (non-copying) and non-erasing). Next, I define, as an example of such a grammar, a simple grammar for Dutch, in which strings are combined by a technique quite similar to Pollard's head wrapping. The main part of the chapter is section 4.4, in which a parsing strategy for linear and non-erasing grammars is proposed. Most 'standard' parsing algorithms for constraint-based grammars (Matsumoto *et al.*, 1983; Pereira and Shieber, 1987;

Shieber, 1989; Haas, 1989; Gerdemann, 1991) are not applicable in general for non-concatenative grammars because in these algorithms the assumption that phrases are constructed by concatenation is 'built-in'. I describe a head-driven parsing algorithm, based on the head-driven parser by Martin Kay (Kay, 1989). The parser is generalized in order to be applicable to any grammar that employs linear and non-erasing operations on strings. The disadvantages Kay noted for his parser do not carry over to this generalized version, as redundant search paths for CF-based grammars turn out to be genuine parts of the search space for this enlarged class of grammars.

The algorithm is closely related to head-driven generators as discussed in the previous chapter. The algorithm proceeds in a bottom-up, head-driven fashion, which provides for bottom-up and top-down filtering in a simple and straightforward way. In modern linguistic theories very much information is defined in lexical entries, whereas rules are reduced to very general (and very un-informative) schemata. More information usually implies a reduction of the search space, hence it is sensible to parse bottom-up, in order to obtain useful information as soon as possible. Furthermore, in many linguistic theories, a 'head' of a construction plays an important role. For example, heads of a construction determine what other parts the construction may have. Furthermore, heads carry the features associated with the construction as a whole (such as case, agreement). The notion 'head' plays an important role in grammatical theories as diverse as Government and Binding, (Xbar theory, Jackendoff (1977)); Generalized Phrase Structure Grammar (the head-feature convention, Gazdar *et al.* (1985)); and Head-driven Phrase Structure Grammar, where the name of the theory reflects the importance of the notion 'head'. Given the importance of the notion 'head', it is sensible to start with the head, in order to know what else you have to look for next. As the parser proceeds from head to head it is furthermore possible to use powerful top-down predictions based on the usual head feature percolations.

In section 4.5 I show how the head-driven parser can be put to use for another instantiation of constraint-based grammars in which string operations are restricted to be linear and non-erasing: constraint-based and lexical versions of Tree Adjoining Grammars.

Some of the properties and possible modifications of the head-corner parser are discussed in section 4.6.

4.2 Beyond concatenation

In formalisms such as PATR II the string associated with a derivation is the sequence of terminal nodes of the corresponding derivation tree in left-to-right order. For example, the sentence

(1) Kim is easy to please

may be analyzed in some PATR grammar in a way that gives rise to the derivation tree in figure 4.1. Hence, the string associated with the derivation is the sequence 'kim is easy to please'. Note though that in PATR this string is not (necessarily) part of the feature structures.

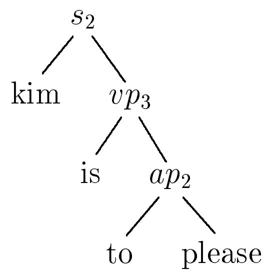


Figure 4.1: A possible PATR derivation tree, for the sentence 'Kim is easy to please'

In sign-based approaches such as in UCG and HPSG the string is part of an attribute of each feature structure (sign). The attribute is usually called 'phon', 'string', 'graph' or 'orth' (I will use 'phon' in the following). Hence, the string associated with a construction is simply the value of the 'phon' feature of the sign that is assigned to the construction. In UCG there is a condition, called 'adjacency', which says that signs can combine only if they are adjacent. In other words, the value of the 'string' feature of a mother node in a parse tree is always the concatenation of the 'string' features of the daughter nodes. Hence, the UCG parse tree for the foregoing example presumably would be something like figure 4.2.

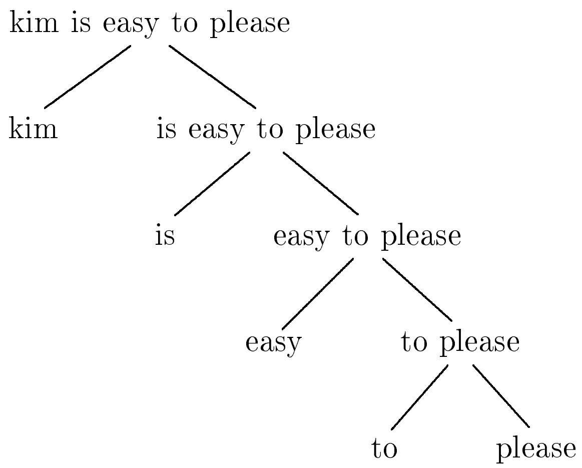


Figure 4.2: Possible UCG parse tree, restricted to the value of the 'phon' attribute, for the sentence 'Kim is easy to please'

The two approaches are formally equivalent, but the second approach has the advantage that it at least becomes easier to think of other 'modes' of combination of the value of the 'phon' attribute. As an example consider

(2) Kim is an easy person to please

Suppose that there is linguistic motivation that in this sentence, as in sentence (1), the sequence 'easy to please' should be regarded as a (discontinuous) constituent. Such an analysis cannot be defined directly in PATR or UCG. If no adjacency condition applied we could have a parse tree of 'easy person to please' as in figure 4.3.

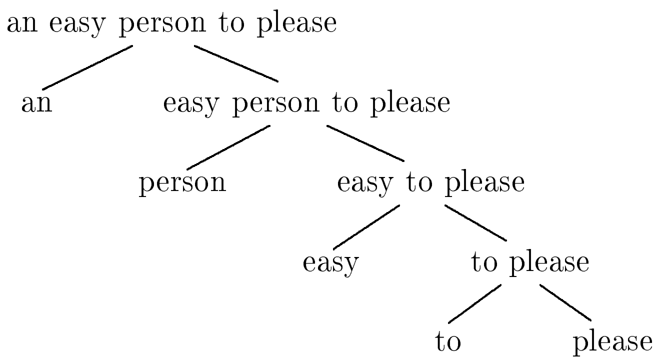


Figure 4.3: Hypothetical parse-tree (restricted to phonological information), for a discontinuous analysis of the sentence 'An easy person to please'

In the next subsections I describe some proposals which allow such a direct implementation of discontinuous constituents.

4.2.1 Head wrapping

Pollard (1984) proposes a grammatical formalism called Head Grammar (HG). HG is a slightly more powerful formalism than context-free grammar. The extra power is available through *head wrapping* operations. A head wrapping operation manipulates strings which contain a distinguished element (its *head*). Such *headed strings* are a pair of an ordinary string, and an index (the pointer to the head), for example $\langle w_1 w_2 w_3 w_4, 3\rangle$ is the string $w_1 w_2 w_3 w_4$ of which the head is w_3. Grammar rules define operations on such strings. Such an operation takes n headed strings as its arguments and returns a headed string. A simple example is the operation which takes two headed strings and concatenates the first one to the left of the second one, and where the head of the first one is the head of the result (this shows that the operations subsume ordinary concatenation). The rule is labelled LC1 by Pollard:

$$LC1(\langle \sigma, i\rangle, \langle \tau, j\rangle) := \langle \sigma\tau, i\rangle$$

The following example shows that head-wrapping operations are in general more powerful than concatenation. In this example the second argument is 'wrapped' around the first argument:

$$RL2(\langle \sigma, j \rangle, \langle t_1 \ldots t_n, i \rangle) := \langle t_1 \ldots t_i \sigma t_{i+1} \ldots t_n, i \rangle$$

As an example, Pollard presents a rule for English auxiliary inversion:

$$S[+INV] \rightarrow RL2(NP, VP[+AUX])$$

which may combine the noun phrase 'Kim' and the verb phrase 'must go' to yield 'Must Kim go', with head 'must'.

The motivation Pollard presents for extending context-free grammars (in fact, GPSG), is of a linguistic nature. Especially so-called discontinuous constituencies can be handled by HG whereas they constitute typical puzzles for GPSG. Apart from the above mentioned subject-auxiliary inversion he discusses the analysis of 'transitive verb phrases' based on Bach (1979). The idea is that in sentences such as

(3) Sandy persuaded Kim to leave

'persuaded' + 'to leave' form a (VP) constituent, which then combines with the NP object ('Kim') by a wrapping operation.

Yet another example of the use of head-wrapping in English are the analyses of the following sentences.

(4) a. Kim is much taller than Sandy
   b. Kim is a much taller person than Sandy

(5) a. Kim is very easy to please
   b. Kim is a very easy person to please

where in the first two cases 'taller than Sandy' is a constituent, and in the latter examples 'easy to please' is a constituent.

Breton and Irish are VSO languages, for which it has been claimed that the V and the O form a constituent. Such an analysis is readily available using head wrapping, thus providing a non-transformational alternative for the analysis of McCloskey (1983).

Finally, Pollard also provides a wrapping analysis of Dutch cross-serial dependencies.

4.2.2 Johnson's 'combines'

Johnson (1985) discusses an extension of DCG in order to analyze the Australian free word-order language 'Guugu Yimidhirr'. In ordinary DCG a category is associated with a pair indicating which location the constituent occupies. Johnson proposes that constituents in the extended version of DCG be associated with a set of such pairs. A constituent thus 'occupies' a set of continuous locations. The following is a sentence of Guugu Yimidhirr:

(6) Yarraga-aga-mu-n gudaa dunda-y biiba-ngun
boy-GEN-mu-ERG do+ABS hit-PAST father-ERG
The boy's father hit the dog

In this sentence, the discontinuous constituent 'Yarraga-aga-mu-n ... biiba-ngun' (boy's father) is associated with the set of locations:

$$\{[0,1],[3,4]\}$$

Johnson notices that such expressions can be represented with bit vectors. In a grammar rule the sets of locations of the daughters of the rule are 'combined' to construct the set of locations associated with the mother node. The predicate $combines(s_1, s_2, s)$ is true iff s is equal to the (bit-wise) union of s_1 and s_2, and the (bit-wise) intersection of s_1 and s_2 is null (ie. s_1 and s_2 must be non-overlapping locations). Grammars which exclusively use this predicate, are permutation-closed. For Guugu Yimidhirr Johnson also proposes a concatenative rule for possessive noun constructions in which the possessive is identified by position rather than by inflectional markings. Apart from these constructions Guugu Yimidhirr is said to be permutation-closed (Johnson, quoting Haviland (1979)).

Earley deduction (Pereira and Warren, 1983) is used as a general proof procedure for such extended DCG grammars.

4.2.3 Sequence union

Reape (1989), and Reape (1990) discuss a relation called *sequence union* to analyze discontinuous constituents. The sequence union of the sequences s_1, s_2 and s_3 is true, iff each of the elements in s_1 and s_2 occur in s_3, and moreover, the original order of the elements in s_1 and s_2 is preserved in s_3. For example, the sequence union of the sequences $\langle a, b\rangle$ and $\langle c, d\rangle$ and s_3 is true, iff s_3 is any of the sequences:

$\langle a, b, c, d\rangle$
$\langle a, c, b, d\rangle$
$\langle a, c, d, b\rangle$
$\langle c, d, a, b\rangle$
$\langle c, a, d, b\rangle$
$\langle c, a, b, d\rangle$

Reape presents an HPSG-style grammar (Pollard and Sag, 1987) for German and Dutch which uses the sequence union relation on *word-order domains*. The grammar handles several word-order phenomena in German and Dutch. Word-order domains are sequences of *signs*. The phonology of a sign is the concatenation of the phonology of the elements of its word-order domain. In 'rules', the word-order domain of the mother sign is defined in terms of the word-order domains of its daughter signs. For example, in the ordinary case the word-order domain of the mother simply consists of its daughter signs. Thus, for example, in the rule $s \rightarrow np\ vp$ the word-order domain associated with s would consist of the sequence which consists of the signs associated with np and vp. However, in specific cases it is also possible that the word-order domain of the mother consists of the elements of the word-order domains of the daughters. Thus, in case of a rule $x \rightarrow yz$ in which the word-order domain associated with y is the sequence $\langle y_1, y_2 \rangle$, and the word-order domain associated with z is $\langle z_1 \rangle$, then the word-order domain associated with x is any of the sequences $\langle y_1, y_2, z \rangle$, $\langle y_1, z, y_2 \rangle$, $\langle z, y_1, y_2 \rangle$.

The following German example by Reape clarifies the approach, where I use indices to indicate to which verb an object belongs.

(7) …es$_i$ ihm$_j$ jemand$_k$ zu lesen$_i$ versprochen$_j$ hat$_k$
…it him someone to read promised had
…*someone had promised him to read it*

It is assumed that a 'flat' verb phrase rule selects the arguments of a verb (in one go), and that furthermore in case of a vp argument, the word-order domain of this vp is sequence-unioned with the word-order domain of the verb; the non-vp arguments of the verb become simply members of the word-order domain of the mother of this verb-phrase rule. Figure 4.4 shows a parse tree of this sentence, where the nodes of the derivation tree are labelled by the string associated with that node. Note that strings are defined *with respect to* word-order domains. Sequence union is defined on such domains. The strings of the derivation tree are thus only indirectly related through the corresponding word-order domains. Linear precedence statements are defined with respect to word-order domains. These statements can be thought of either as well-formedness conditions on totally ordered sequences, or alternatively as constraints limiting possible orders of a word-order domain. Note that order information is monotonic; the sequence union relation cannot 'change' the order of two ordered items.

4.2.4 Tree Adjoining Grammars

Tree Adjoining Grammar (TAG) is a formalism originally proposed in Joshi *et al.* (1975). Several variations on that formalism are developed, among which we will be interested in lexicalized (LTAG) (Abeille, 1988; Schabes, 1990) and constraint-based (FTAG) (Vijay-Shanker and Joshi, 1988; Vijay-Shanker, to appear) versions. A TAG consists of a number of elementary trees, which can be combined with a substitution, and an adjunction operation.

The TAG formalism can be motivated because it provides a larger domain of locality in which to state linguistic dependencies. In most formalisms, dependencies can be defined between the elements of a rule (i.e. between the nodes of a local tree). In TAG, it is possible to state dependencies between nodes of trees which are further apart, because the basic building blocks of the formalism are trees. For example, the relation between a topicalized constituent and its governor can be stated locally in a TAG, whereas in most other formalisms this relation must be stated with some global mechanism. Furthermore, the adjunction operation provides for the analysis of (at

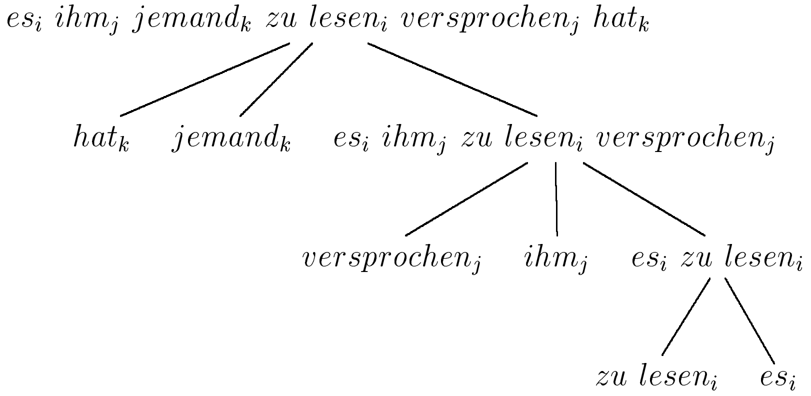


Figure 4.4: Simplified parse tree of Reape's analysis of German verb-clusters, using sequence union. The indices relate verbs with their arguments. This tree can be read in a bottom-up fashion, as follows. Firstly, the verb 'zu lesen' selects an np, resulting in a verb-phrase with word-order domain $\langle np_i, v_i \rangle$. This verb-phrase is one of the arguments of the verb 'versprochen'. As this argument is a verb-phrase its word-order domain is sequence-unioned. A (possible) result is the word-order domain $\langle np_i, np_j, v_i, v_j \rangle$. The verb 'hat' also selects an np and a vp. The elements of the vp are sequence unioned, the np is simply added to the word-order domain, which results in a possible word-order domain $\langle np_i, np_j, np_k, v_i, v_j, v_k \rangle$.

least some) kinds of discontinuous constituency constructions. This is reflected in the formal power of the formalism: some TAGs recognize context-sensitive languages.

Ordinary TAG. A Tree Adjoining Grammar consists of a set of *elementary trees*, divided in *initial* and *auxiliary* trees. These trees constitute the basic building blocks of the formalism. Operations of *adjunction* and *substitution* are defined which build *derived trees* from elementary trees. TAG thus constitute a tree-generating system, rather than a string-generating system as context-free grammar. The string language of a TAG is defined as the yields of all the trees derived by a TAG.

An initial tree is a tree of which the interior nodes are all labelled with non-terminal symbols, and the nodes on the frontier are either labelled with terminal symbols, or with non-terminal symbols, which are marked with the substitution marker (↓).

An auxiliary tree is defined as an initial tree, except that exactly one of its frontier nodes must be marked as *foot node* (‘*’). The foot node must be labelled with a non-terminal symbol which is the same as the label of the root node.

As an example, consider the following initial and auxiliary trees in figure 4.5. In

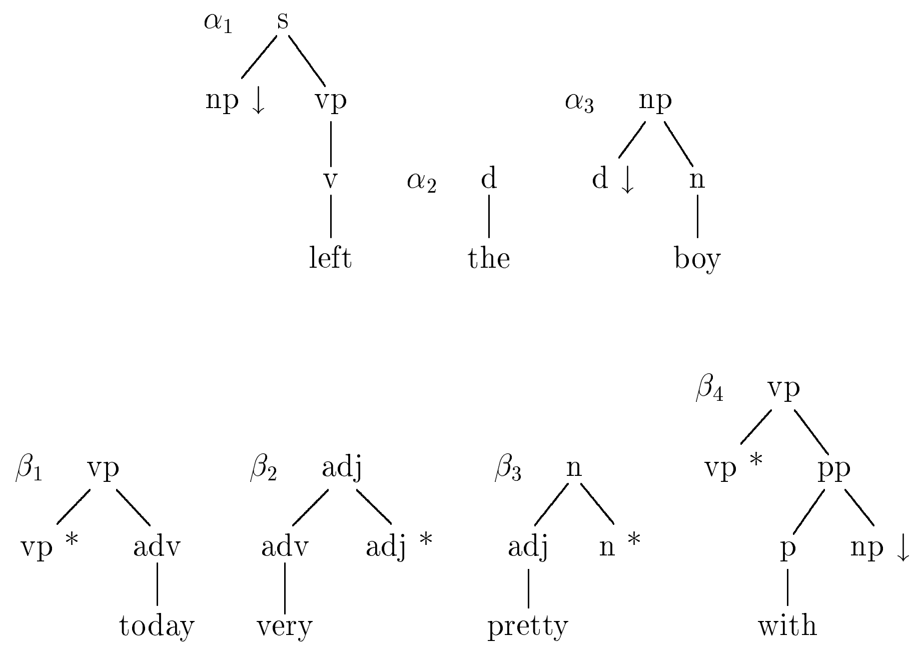
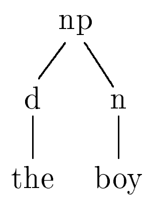


Figure 4.5: Three initial and four auxiliary trees as an example of a Tree Adjoining Grammar.

initial tree α_1, the `np` node is a substitution node, and `left` is a terminal symbol associated with a frontier node. In auxiliary tree β_4 the foot node is the leftmost daughter of the root node. `with` is a terminal symbol at a frontier node, and the `np` node is a substitution node.

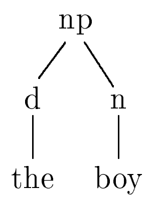
Derived trees are built from initial and auxiliary trees by *substitution* and *adjunction*. Substituting a tree α in a tree α' simply replaces a substitution node in α' with α, under the convention that the non-terminal symbol of the substitution node is the same as the root node of α. For example, substituting α_2 in α_3 gives the following tree:



Only initial trees, and derived trees, can be substituted in another tree.

Adjunction is a more complex operation. Adjoining an auxiliary tree β at some node n of a derived tree γ proceeds as follows. Firstly, the non-terminal symbol of the root node (and hence the non-terminal symbol of the foot node) of β should be the same as the non-terminal symbol associated with n. The sub-tree t of γ rooted by n is removed from γ, and β is substituted for it instead; where t is substituted in the foot node of β. An illustration of the adjunction operation is presented in figure 4.6.

As an example, consider the adjunction of β_3 at the 'n' node of the derived tree:



This yields the derived tree in figure 4.7. As a further example, we might then adjoin β_2 into this derived tree at the `adj` node, resulting in the tree shown in figure 4.8.

Adjoining constraints. Usually, a TAG may specify *adjoining constraints* on the nodes of its initial and auxiliary trees. Such constraints for example may explicitly forbid adjunction on a node, or only allow adjunction of certain auxiliary trees. However, we will ignore these adjoining constraints because in FTAG these can all be simulated using feature equations.

Lexicalized TAG. In LTAG, each elementary tree contains at least one frontier node labelled with a terminal symbol. Thus each elementary tree is associated with at least one lexical element. Note that in the example grammar above each of the elementary trees is lexical; hence the grammar provides an example of a lexicalized TAG.

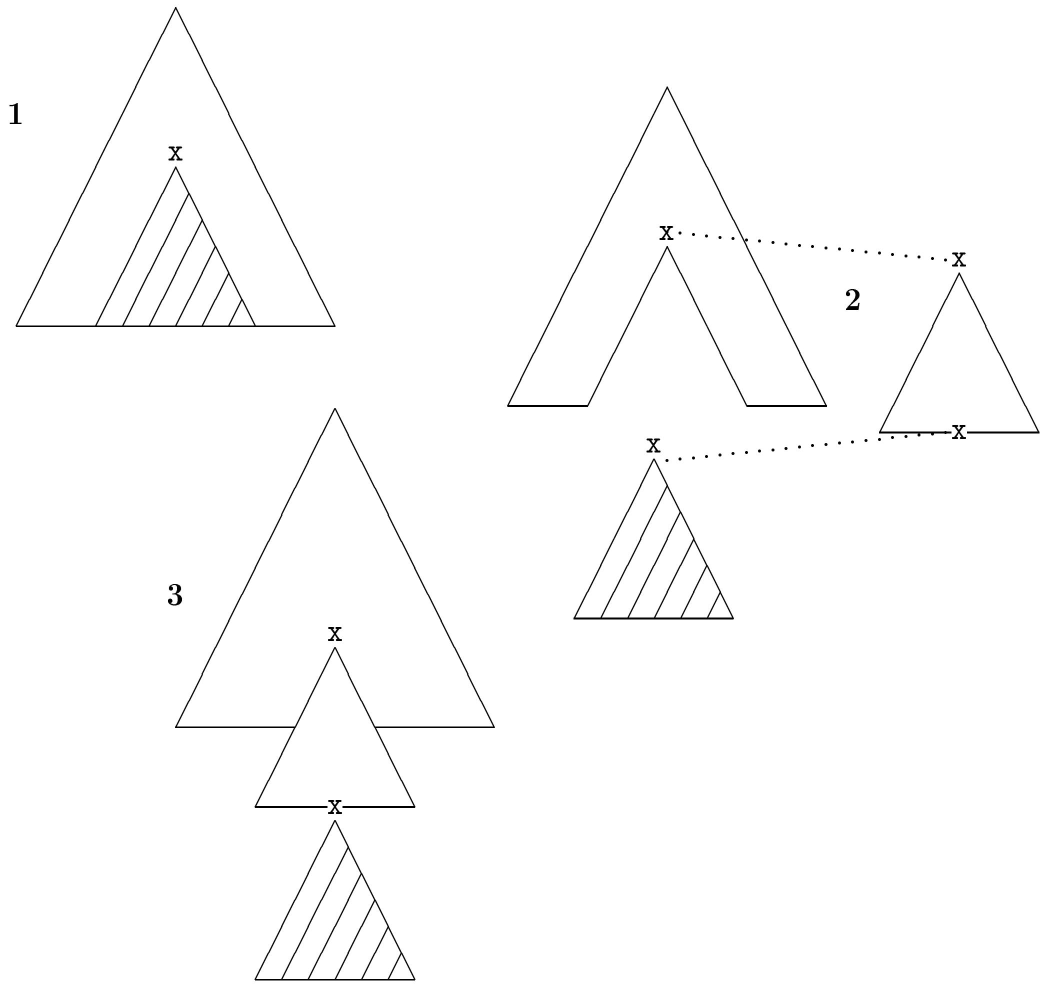


Figure 4.6: Illustration of adjunction. In (1) the sub-tree at the node n where adjunction takes place, is identified. In (2) the root node of the auxiliary tree is substituted at node n, and the sub-tree of n is substituted in the foot node of the auxiliary tree, resulting in (3).

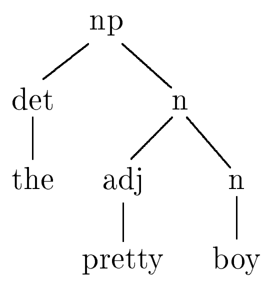


Figure 4.7: Result of adjoining β_3 at the node labelled 'n', of the derived tree np(d(the), n(boy)).

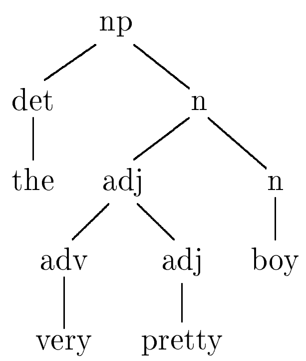


Figure 4.8: Result of adjoining β_2 at the 'adj' node of the derived tree np(d(the), n(adj(pretty), n(boy))).

Constraint-based TAG. Adding constraints to the TAG formalism may at first sight be somewhat similar to the addition of constraints to a context-free grammar: each non-terminal symbol may be constrained by equations. However, in the case of a TAG this will not do. If we simply would constrain a node of an elementary tree, and then perform adjunctions at this node, then this would lead to a situation where these constraints are associated both to two distinct nodes.

A way to look at this problem from a more general perspective is presented in Vijay-Shanker (to appear). In a way, an elementary tree *partially describes* possible trees. For example, the elementary tree α_1 of figure 4.5 can be interpreted as in figure 4.9.

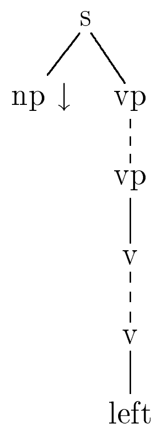


Figure 4.9: Dominance relations in TAG. Solid lines represent immediate dominance, and dotted lines any dominance.

Thus, we might view such an initial tree as licensing any tree which can be built by adjoining at its interior nodes. For that reason, there is always a sense in which we view such nodes 'from the bottom' or 'from the top'. This intuition is formalized in constraint-based versions as follows. Each of the nodes of an elementary tree is associated with two variables (called *top* and *bot*) representing the view 'from the top' and 'from the bottom' on which constraints can be defined.

In the case of the substitution of a tree α, at a substitution node n of a tree α', the result is as before, under the convention that the *top* variable of the root node of α is constrained to be equal to the *top* variable of n, and similarly the *bot* variable of the root of α is equal to the *bot* variable of n.

Adjunction is slightly more complex. Adjoining an auxiliary tree β at some node n of a derived tree γ is defined as before, under the convention that the *top* variable of n is constrained to be equal to the *top* variable of the root node of β; furthermore the *bot* variable of n is constrained to be equal to the *bot* variable of the foot node of β. See figure 4.10 for an illustration.

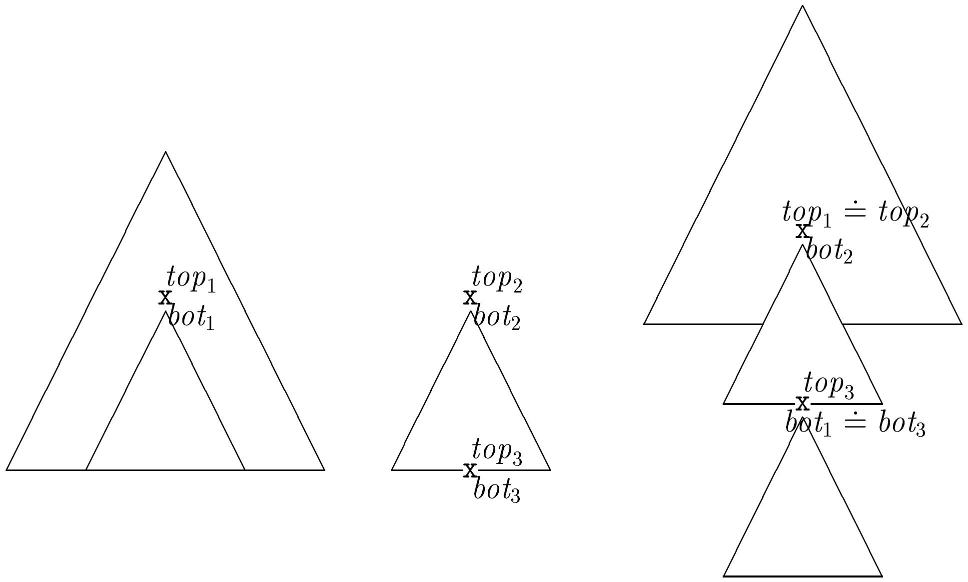


Figure 4.10: Illustration of adjunction with constraints. The top features of the adjunction node are unified with the top features of the root node of the auxiliary tree. The bottom features of the adjunction node are unified with the bottom features of the foot node of the auxiliary tree.

Finally, at the end of a derivation the *top* and *bot* variables of each node are constrained to be equal as well. A derived tree with a node of which the *top* and *bot* variables cannot be unified is not regarded as a tree derived by the grammar.

Examples of the use of constraints for TAG are the usual number agreement, case marking, etc. However, it is also possible to use the constraint machinery to define 'sign-based' TAGs in which each node is associated with a value for a 'phonology' and a 'semantics' attribute. We shall see that this will crucially illustrate the need for two distinct variables for each node.

The figures 4.11 and 4.12 illustrate how this might work. The phonology associated with the bottom part of the verb is the sequence $\langle saw \rangle$. However, adjunctions may apply to this node. For that reason, the phonology of the top part of this node is not yet known. Whatever its value will become, we know that the phonology of the bottom part of the verb phrase node, is the concatenation of the top parts of the verb node and the object np. Again, adjunctions at the verb phrase node may apply. Therefore the bottom and top parts of the verb phrase node are unrelated. The phonology of the bottom of the root node is the concatenation of the top of the subject np and the verb phrase. Note that, if no adjunctions apply at all, the bottom and top parts of each node is unified, giving exactly the right result.

Thus, the argument structure of the bottom part of the verb 'saw' is defined as a binary predicate, of which the arguments are the argument structures of the subject and the object. The argument structure of the top part of this verb, however, is not yet known. For example, modifiers may adjoin later at this node to construct a more complex argument structure, built from the simple one associated with the bottom part of the node. Once no more adjunctions are applied, the argument structure is

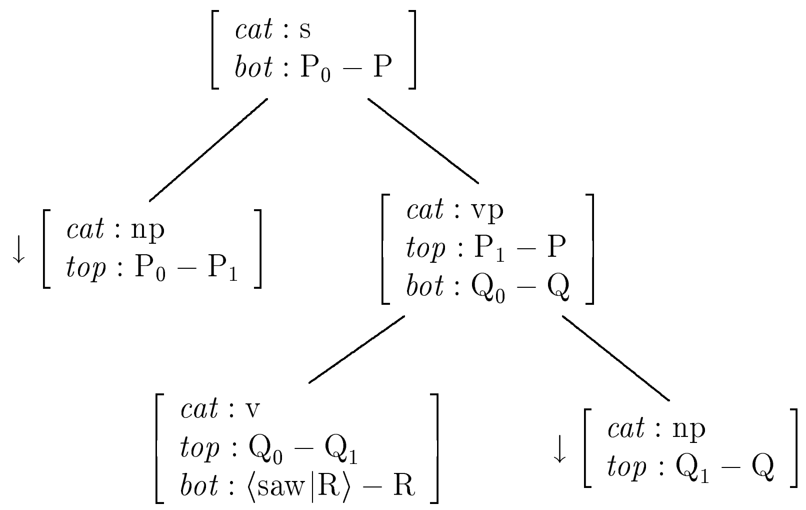


Figure 4.11: Example of sign-based TAG: construction of phonology, using a difference-list encoding. Each local tree defines that the bottom string of the mother is the concatenation of the top of the daughters. However, the bottom and top of each node are not equated, as adjunctions may take place at each node.

percolated to the verb phrase node: hence the top part of the verb is unified with the bottom part of the verb-phrase. Again, the bottom-part of the verb phrase may be modified, because of adjunctions. Its top-part is unified with the bottom part of the root node, because the argument structure should be percolated, if no more adjunctions apply.

4.2.5 Linear and non-erasing grammars

I will restrict the attention to a class of constraint-based formalisms, in which operations on strings are defined that are more powerful than concatenation, but which operations are restricted to be *non-erasing*, and *linear*. The resulting class of systems has an obvious relation to Linear Context-Free Rewriting Systems (LCFRS), (Vijay-Shanker *et al.*, 1987), except for the addition of constraints. For a discussion of the properties of LCFRS without feature-structures, see Vijay-Shanker *et al.* (1987) and Weir (1988). We will not discuss these properties, as these properties do not carry over to the current system. The proposals discussed in the previous section can be seen as examples of linear and non-erasing constraint-based grammars.

As in LCFRS, the operations on strings are characterized as follows. First, derived structures will be mapped onto a string of words; i.e. each derived structure 'knows' which string it 'dominates'. For example, each derived feature structure may contain an attribute 'string' of which the value is a list of atoms representing the string it

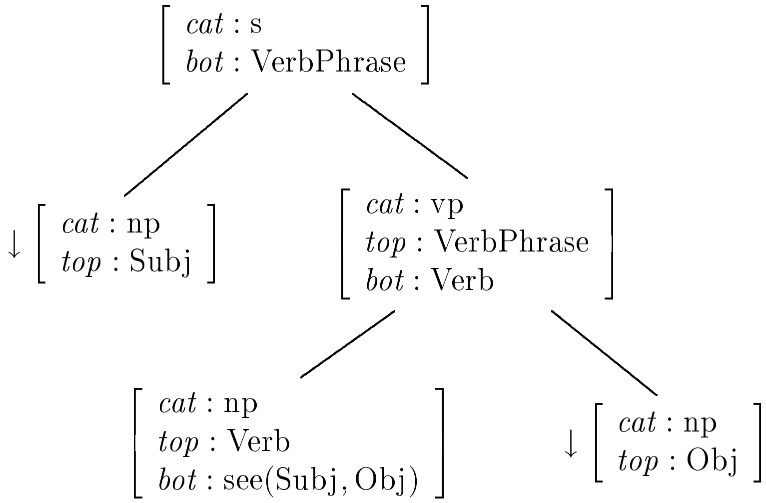


Figure 4.12: Example of a sign-based TAG: construction of semantics. The bottom and top semantic structures of each node are not equated, as adjunctions may take place at each node.

dominates. I will write $w(F)$ for the set of occurrences of words that the derived structure F dominates. Rules combine structures $D_1 \ldots D_n$ into a new structure M. Non-erasure requires that the union of w applied to each daughter is a subset of $w(M)$:

$$\bigcup_{i=1}^{n} w(D_i) \subseteq w(M)$$

Linearity requires that, for each rule, the difference of the cardinalities of these sets is a constant factor; i.e. a rule may only introduce a fixed number of words syncategorematically:

$$|w(M)| - |\bigcup_{i=1}^{n} w(D_i)| = c, c \textit{ a constant}$$

CF-based formalisms clearly fulfill this requirement, as do Head Grammars, grammars using sequence union, Johnson's Australian grammar, and TAG's. Unlike in the definition of LCFRS I do not require that these operations are operations in the strict sense, i.e. the operations do not have to be functional, nor do I require that these operations are defined as operations on sequences of strings. Note that sequence union is relational; the others are functional. For a discussion of the consequences of this difference, cf. Reape (1991).

Furthermore, I will assume that some rules have a designated daughter, called the head. Although I will not impose any restrictions on the head, it will turn out that the parsing strategy to be proposed will be very sensitive to the choice of heads, with the effect that grammars in which the notion 'head' is defined in a systematic way (Pollard's Head Grammars, Reape's version of HPSG, Dowty's version of Categorial Grammar), may be much more efficiently parsed than other grammars. The notion *head corner* of a parse tree is defined recursively in terms of the head. The head-corner of a node without a head (eg. a frontier node) will be this node itself. The head corner of a node with a head, will be the head corner of this head node.

Remember that a grammar is a set of definite clauses, defining the relation 'sign'. Hence a grammar rule will be something like

$$sign(\mathrm{M})\,{:}{-}\,sign(\mathrm{D}_1)\ldots sign(\mathrm{D}_n), \phi.$$

where ϕ are constraints on the variables. However, to make the ways in which strings are combined explicit, I will allow that a rule is associated with an extra relation-call $cp/2$ (for 'construct-phonology), of which the arguments are resp. the mother node, and the sequence of daughter nodes. This extra relation is thought of as defining how the string of the mother node is built from the strings of the daughters.

For example, to implement a simple grammar rule using concatenation, we may write

(8) $sign(\mathrm{M})\,{:}{-}$
$\quad sign(\mathrm{D}_1),$
$\quad sign(\mathrm{D}_2),$
$\quad cp(\mathrm{M}, \langle \mathrm{D}_1, \mathrm{D}_2 \rangle),$
$\quad \phi.$

$$cp(\left[\ phon : \mathrm{P}\ \right], \langle \left[\ phon : \mathrm{P}_1\ \right], \left[\ phon : \mathrm{P}_2\ \right] \rangle)\,{:}{-}$$
$$append(\mathrm{P}_1, \mathrm{P}_2, \mathrm{P}).$$

where the relation 'cp' states that the string of the mother is the concatenation of the strings of the daughters. The extra relation should be defined in such a way that given the instantiated daughter signs, the extra relation is guaranteed to terminate (assuming the procedural semantics defined in chapter 2). Thus, we think of the definition of such extra predicates as an (innocent) sub-program.

Furthermore, each grammar should provide a definition of the predicate *head*/2 and *yield*/2. The first one defines the relation between the mother sign of a rule and its head; the second one defines the phonology (as a list of atoms) associated with a sign.

Finally, for each rule it should be known which terminals are introduced by it.

Representing rules for meta-interpreter For the meta-interpreter we define here we will assume that rules are represented as follows. Firstly, as in DCG, I use the convention that external calls are written inside { }; furthermore, the meta-interpreter uses the meta-call *call*/1 to call a predicate (as in Prolog).

A rule $sign(\mathrm{Mother})\,{:}{-}\,sign(\mathrm{D}_1)\ldots sign(\mathrm{D}_n), cp(\ldots), \phi.$ which consumes words Words, and which has no head, is called a 'non-chain-rule', and is represented as follows:

(9) $ncr(\text{Mother}, [\text{D}_1 \ldots \text{D}_n, \{\ cp(\ldots)\ \}], \text{Words})\,\texttt{:-}\,\phi.$

Note that lexical entries are non-chain-rules of which the list of daughters is presumably the empty list. Also note that the call to the relation 'cp' is added to the list of daughters.

A rule $sign(\text{Mother})\,\texttt{:-}\,sign(\text{D}_1)\ldots sign(\text{D}_i), sign(\text{Head}), sign(\text{D}_{i+1})\ldots \text{D}_n, cp(\ldots), \phi.$ with head Head, and which consumes words Words, is called a 'chain-rule', and is represented for the meta-interpreter as follows (hence we allow for terminal symbols to be introduced syncategorematically):

(10) $cr(\text{Head}, \text{Mother}, [\text{D}_1 \ldots \text{D}_n, \{\ cp(\ldots)\ \}], \text{Words})\,\texttt{:-}\,\phi.$

4.3 A sample grammar

In this section I present a simple linear and non-erasing constraint-based grammar for a (tiny) fragment of Dutch. As a caveat I want to stress that the purpose of the current section is to provide an *example* of possible input for the parser to be defined in the next section, rather than to provide an account that is completely satisfactory from a linguistic point of view.

There is only one parameterized, binary branching, and headed rule in the grammar. The rule does not introduce any terminals. It is defined as follows, where the first daughter represents the head:

(11)
$$\begin{aligned}
&sign(\begin{bmatrix} syn : \text{Syn} \\ sc : \text{Tail} \\ sem : \text{Sem} \end{bmatrix}_{\text{M}}\ \texttt{:-} \\
&\qquad sign(\begin{bmatrix} syn : \text{Syn} \\ sc : \langle \text{Arg}|\text{Tail}\rangle \\ sem : \text{Sem} \end{bmatrix}_{\text{H}}), \\
&\qquad sign(\text{Arg}), \\
&\qquad cp(\text{M}, \langle \text{H}, \text{Arg}\rangle).
\end{aligned}$$

In this grammar rule, heads select arguments using a subcat list. Argument structures are specified lexically and are percolated from head to head. Syntactic features are shared between heads (hence I make the simplifying assumption that head = functor, which may have to be revised in order to treat modification). The relation 'cp' defines how the string of the mother is constructed from its daughters. In the grammar I use revised versions of Pollard's head wrapping operations to analyze *cross serial dependency* and *verb second* constructions. For a linguistic background of these constructions and analyses, cf. Evers (1975), Koster (1975) and many others. The value of the attribute *h-s* (for 'headed string') consists of three parts, to implement the idea of Pollard's 'headed strings'. The parts *left* and *right* represent the strings left and right of the head. The part *head* represent the head string. Hence, the string associated with such a term is the concatenation of the three arguments from left to right. The predicate *cp* is defined as follows:

(12) $cp(\begin{bmatrix} \textit{h-s} : \text{Mphon} \\ \textit{phon} : \text{String} \end{bmatrix}, \langle \begin{bmatrix} \textit{h-s} : \text{Hphon} \end{bmatrix}, \begin{bmatrix} \textit{h-s} : \text{Aphon} \\ \textit{rule} : \text{Rule} \end{bmatrix} \rangle) \text{:-}$

$\quad wrap(\text{Rule}, \text{Hphon}, \text{Aphon}, \text{Mphon}),$
$\quad phon\_string(\text{Mphon}, \text{String}).$

In the first clause, the values of the attribute *h-s* associated with the two daughters of the rule are to be combined by the *wrap* predicate. Several versions of this predicate will be defined below. The value of *phon* of the mother node is defined with respect to its *h-s* value by the predicate *phon_string*. This predicate is defined in terms of the predicate *append*/3. As an abbreviation I write $\text{A} \cdot \text{B}$ for C such that *append*(A, B, C). The definition of *phon_string* is:

(13) $phon\_string(\begin{bmatrix} \textit{left} : \text{L} \\ \textit{head} : \text{H} \\ \textit{right} : \text{R} \end{bmatrix}, \text{L} \cdot \text{H} \cdot \text{R}).$

A few versions of the predicate *wrap* are listed below, to illustrate the idea that different string operations can be defined. Each version of the predicate will be associated with an atomic identifier to allow lexical entries to subcategorize for their arguments under the condition that a specific version of this predicate be used. The purpose of this feature is similar to the 'order' feature found in UCG (Zeevat *et al.*, 1987). For example, a verb may select an object to its left, and an infinite verb phrase which has to be raised. For simple (left or right) concatenation the predicate is defined as follows:

(14) $wrap(\text{left}, \begin{bmatrix} \textit{left} : \text{L} \\ \textit{head} : \text{H} \\ \textit{right} : \text{R} \end{bmatrix}, \begin{bmatrix} \textit{left} : \text{AL} \\ \textit{head} : \text{AH} \\ \textit{right} : \text{AR} \end{bmatrix}, \begin{bmatrix} \textit{left} : \text{AL} \cdot \text{AH} \cdot \text{AR} \cdot \text{L} \\ \textit{head} : \text{H} \\ \textit{right} : \text{R} \end{bmatrix}).$

$wrap(\text{right}, \begin{bmatrix} \textit{left} : \text{L} \\ \textit{head} : \text{H} \\ \textit{right} : \text{R} \end{bmatrix}, \begin{bmatrix} \textit{left} : \text{AL} \\ \textit{head} : \text{AH} \\ \textit{right} : \text{AR} \end{bmatrix}, \begin{bmatrix} \textit{left} : \text{L} \\ \textit{head} : \text{H} \\ \textit{right} : \text{R} \cdot \text{AL} \cdot \text{AH} \cdot \text{AR} \end{bmatrix}).$

In the first case the string associated with the argument is appended to the left of the string left of the head; in the second case this string is appended to the right of the string right of the head.

Lexical entries for intransitive verbs such as 'ontwaakt' (wakes up) are defined as follows:

$$\text{(15)}\quad \textit{sign}\left(\left[\begin{array}{l} \textit{syn} : \mathrm{v} \\ \textit{sc} : \langle \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Subj} \\ \textit{rule} : \mathrm{left} \end{array}\right] \rangle \\ \textit{h-s} : \left[\begin{array}{l} \textit{left} : \langle\rangle \\ \textit{head} : \langle \mathrm{ontwaakt} \rangle \\ \textit{right} : \langle\rangle \end{array}\right] \\ \textit{sem} : \mathrm{ontwaakt(Subj)} \\ \textit{phon} : \langle \mathrm{ontwaakt} \rangle \end{array}\right]\right).$$

I assume that lexical entries also specify that their *phon*-value is dependent on the *h-s* value. Furthermore, the values of the *left* and *right* attributes of *h-s* are the empty list. Henceforth, I will not specify the values of *phon* and *h-s* explicitly, but assume that each lexical entry extends

$$\left[\begin{array}{l} \textit{h-s} : \left[\begin{array}{l} \textit{left} : \langle\rangle \\ \textit{head} : \mathrm{Head} \\ \textit{right} : \langle\rangle \end{array}\right] \\ \textit{phon} : \mathrm{Head} \end{array}\right].$$

Hence, bi-transitive verbs such as ‘vertelt’ (tells) are abbreviated as follows:

$$\text{(16)}\quad \left[\begin{array}{l} \textit{syn} : \mathrm{v} \\ \textit{sc} : \langle \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Obj} \\ \textit{rule} : \mathrm{left} \end{array}\right], \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Iobj} \\ \textit{rule} : \mathrm{left} \end{array}\right], \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Subj} \\ \textit{rule} : \mathrm{left} \end{array}\right] \rangle \\ \textit{phon} : \langle \mathrm{vertelt} \rangle \\ \textit{sem} : \mathrm{vertelt(Subj,Iobj,Obj)} \end{array}\right]$$

A different version of this lexical entry selects an sbar (complementizer phrase) to the right (simplifying the argument structure):

$$\text{(17)}\quad \left[\begin{array}{l} \textit{syn} : \mathrm{v} \\ \textit{sc} : \langle \left[\begin{array}{l} \textit{syn} : \mathrm{comp} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Obj} \\ \textit{rule} : \mathrm{right} \end{array}\right], \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Iobj} \\ \textit{rule} : \mathrm{left} \end{array}\right], \left[\begin{array}{l} \textit{syn} : \mathrm{n} \\ \textit{sc} : \langle\rangle \\ \textit{sem} : \mathrm{Subj} \\ \textit{rule} : \mathrm{left} \end{array}\right] \rangle \\ \textit{phon} : \langle \mathrm{vertelt} \rangle \\ \textit{sem} : \mathrm{vertelt(Subj,Iobj,Obj)} \end{array}\right]$$

Proper nouns such as ‘Arie’ are simply defined as:

(18) $\begin{bmatrix} \mathit{syn} : \text{n} \\ \mathit{sc} : \langle\rangle \\ \mathit{phon} : \langle\text{arie}\rangle \\ \mathit{sem} : \text{arie} \end{bmatrix}$

For the sake of the example I assume several other NP's to have such a definition.

The choice of data-structure for the value of the attribute *h-s* allows a simple definition of the verb raising *vr* version of the *wrap* predicate that may be used for Dutch cross serial dependencies:

(19) $\mathit{wrap}(\text{vr}, \begin{bmatrix} \mathit{left} : \langle\rangle \\ \mathit{head} : \text{H} \\ \mathit{right} : \langle\rangle \end{bmatrix}, \begin{bmatrix} \mathit{left} : \text{AL} \\ \mathit{head} : \text{AH} \\ \mathit{right} : \text{AR} \end{bmatrix}, \begin{bmatrix} \mathit{left} : \text{AL} \\ \mathit{head} : \text{H} \\ \mathit{right} : \text{AH} \cdot \text{AR} \end{bmatrix}).$

Here the head and right string of the argument are appended to the right, whereas the left string of the argument is appended to the left. A raising verb, eg. 'hoort' (hears) is defined as:

(20) $\begin{bmatrix} \mathit{syn} : \text{v} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{syn} : \text{inf} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{sem} : \text{InfSj} \end{bmatrix} \rangle \\ \mathit{sem} : \text{Oj} \\ \mathit{rule} : \text{vr} \end{bmatrix}, \begin{bmatrix} \mathit{syn} : \text{n} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \text{InfSj} \\ \mathit{rule} : \text{left} \end{bmatrix}, \begin{bmatrix} \mathit{syn} : \text{n} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \text{Sj} \\ \mathit{rule} : \text{left} \end{bmatrix} \rangle \\ \mathit{phon} : \langle\text{hoort}\rangle \\ \mathit{sem} : \text{hoort(Sj,Oj)} \end{bmatrix}$

In this entry 'hoort' selects — apart from its NP-subject — two objects, an NP and a VP (with category INF). The INF still has an element in its subcat list; this element is controlled by the NP (this is performed by the sharing of InfSj). To derive the subordinate phrase

(21) dat Jan Arie Bob leugens hoort vertellen
that Jan Arie Bob lies hears tell
that Jan hears that Arie tells lies to Bob

the main verb 'hoort' first selects the infinitival 'bob leugens vertellen'. These two strings are combined into 'bob leugens hoort vertellen' (using the vr version of the *wrap* predicate). After the selection of the object, resulting in 'arie bob leugens hoort vertellen', the subject is selected resulting in the string 'jan arie bob leugens hoort vertellen'. This string is selected by the complementizer, resulting in 'dat jan arie bob leugens hoort vertellen'. The argument structure will be instantiated as dat(hoort(jan,vertelt(arie,bob,leugens))).

Note that this analysis of verb raising constructions faces problems because of the possibility to coordinate verb clusters. This possibility seems to indicate that an analysis in which subcategorization lists are manipulated (as discussed in the previous chapter) is more promising. For a discussion of these matters, cf. **?**.

In Dutch main clauses, there usually is no overt complementizer; instead the finite verb occupies the first position (in yes-no questions), or the second position (right after the topic; ordinary declarative sentences). In the following analysis an empty complementizer selects an ordinary (finite) vp; the resulting string is formed by the following definition of *wrap*.

(22) $wrap(\mathrm{v2}, \begin{bmatrix} \mathit{left} : \langle\rangle \\ \mathit{head} : \langle\rangle \\ \mathit{right} : \langle\rangle \end{bmatrix}, \begin{bmatrix} \mathit{left} : \mathrm{L} \\ \mathit{head} : \mathrm{H} \\ \mathit{right} : \mathrm{R} \end{bmatrix}, \begin{bmatrix} \mathit{left} : \langle\rangle \\ \mathit{head} : \mathrm{H} \\ \mathit{right} : \mathrm{L} \cdot \mathrm{R} \end{bmatrix}).$

The 'empty' finite complementizer is defined as:

(23) $\begin{bmatrix} \mathit{syn} : \mathrm{comp} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{syn} : \mathrm{v} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{Obj} \\ \mathit{rule} : \mathrm{v2} \end{bmatrix} \rangle \\ \mathit{phon} : \langle\rangle \\ \mathit{sem} : \mathrm{Obj} \end{bmatrix}$

whereas an ordinary complementizer, eg. 'dat' (that) is defined as:

(24) $\begin{bmatrix} \mathit{syn} : \mathrm{comp} \\ \mathit{sc} : \langle \begin{bmatrix} \mathit{syn} : \mathrm{v} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{Obj} \\ \mathit{rule} : \mathrm{right} \end{bmatrix} \rangle \\ \mathit{phon} : \langle \mathit{dat} \rangle \\ \mathit{sem} : \mathrm{Obj} \end{bmatrix}$

Thus, after the application of the empty complementizer, a verb initial sentence is formed. In the case of root sentences, some mechanism for topicalization will apply, which in some way places a further constituent before the verb. In yes-no questions, the derivation is finished at this point.

Note that this analysis captures the special relationship between complementizers and (fronted) finite verbs in Dutch. The sentence

(25) Hoort Arie Jan Bob vertellen dat Claire ontwaakt?
hears Arie Jan Bob tell that Claire wakes up?
Does Arie hear that Jan tells Bob that Claire wakes up?

is derived as in figure 4.13 (where the head of a string is represented in capitals).

What remains to be done is to define the two grammar specific predicates *head*/2 and *yield*/2. These are simply defined as follows:

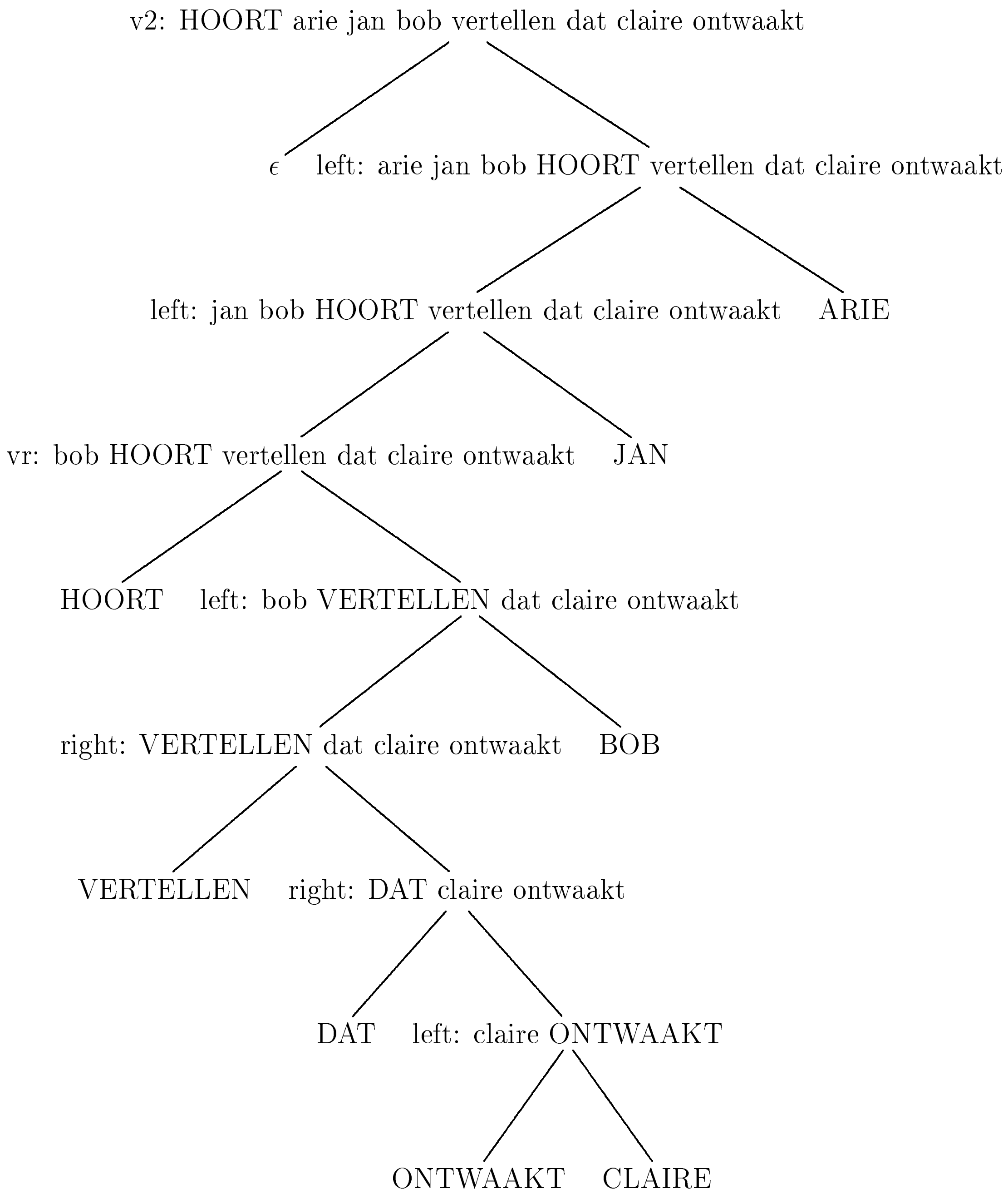


Figure 4.13: Deriving ‘Hoort Arie Jan Bob vertellen dat Claire ontwaakt’

(26) $\mathit{head}(\left[\ \mathit{syn} : \mathrm{Syn}\ \right], \left[\ \mathit{syn} : \mathrm{Syn}\ \right]).$

$\mathit{yield}(\left[\ \mathit{phon} : \mathrm{String}\ \right], \mathrm{String}).$

4.4 The head corner parser

This section describes the head-driven parsing algorithm for the type of grammars described above. The parser is a generalization of Kay's head-driven parser, which in turn is a modification of a left-corner parser. The parser, which may be called a 'head-corner' parser,[1] proceeds in a bottom-up way. Because the parser proceeds from head to head it is easy to use powerful top-down predictions based on the usual head feature percolations, and subcategorization requirements that heads impose upon their arguments. In fact, the motivation for this approach to parsing discontinuous constituency is already hinted at by Mark Johnson (Johnson, 1985):

> My own feeling is that the approach that would bring the most immediate results would be to adopt some of the "head driven" aspects of Pollard's (1984) Head Grammars. In his conception, heads contain as lexical information a list of the items they subcategorize for. This strongly suggests that one should parse according to a "head-first" strategy: when one parses a sentence, one looks for its verb first, and then, based on the lexical form of the verb, one looks for the other arguments in the clause. Not only would such an approach be easy to implement in a DCG framework, but given the empirical fact that the nature of argument NP's in a clause is strongly determined by that clause's verb, it seems a very reasonable thing to do.

It is clear from the context that Johnson believes this strategy especially useful for non-configurational languages.

In left-corner parsers (Matsumoto *et al.*, 1983) the first step of the algorithm is to select the left-most word of a phrase. The parser then proceeds by proving that this word indeed can be the left-corner of the phrase. It does so by selecting a rule of which the leftmost daughter unifies with the category of the word. It then parses other daughters of the rule recursively and then continues by connecting the mother category of that rule upwards, recursively. An illustration of the left-corner parser is provided in figure 4.14.

A head-driven algorithm can be defined analogously (Kay (1989)), if we replace the notion 'left' with 'head'. Thus, to prove a given goal, the parser selects a lexical entry (the head). Then the parser continues to prove that this lexical entry indeed is the head of the goal, by selecting a rule of which this lexical entry can be head. The other daughters of the rule are parsed recursively, and the result constitutes a slightly

[1] This name is due to Pete Whitelock.

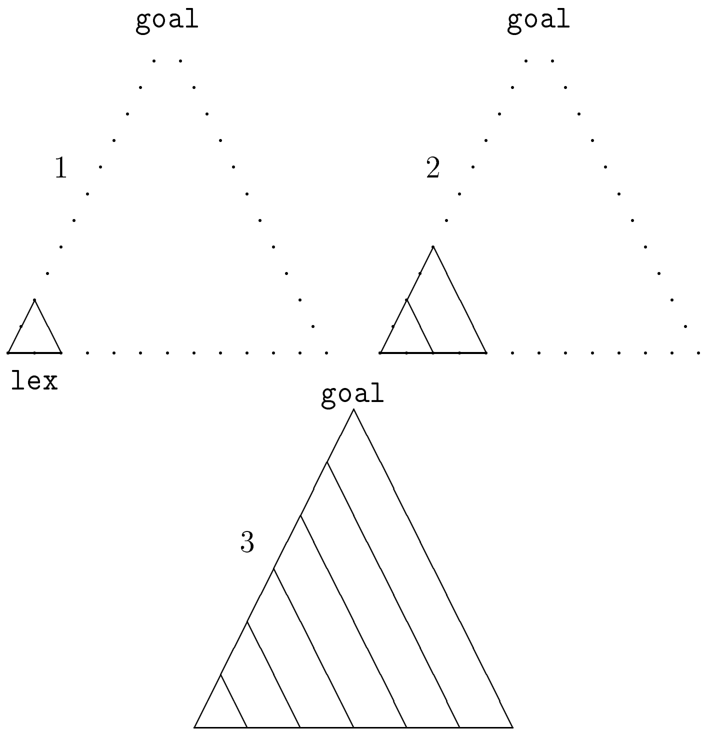


Figure 4.14: The left-corner parser. The parser selects the left-most element of the string (1), and proves that this element is the left-corner of the goal. To this end, a rule is selected of which this lexical entry is the left-most daughter. The other daughters of the rule are parsed recursively. The result is a slightly larger left-corner (2). This process repeats itself until a left-corner is constructed which dominates the whole string (3). The string is recognized from left to right.

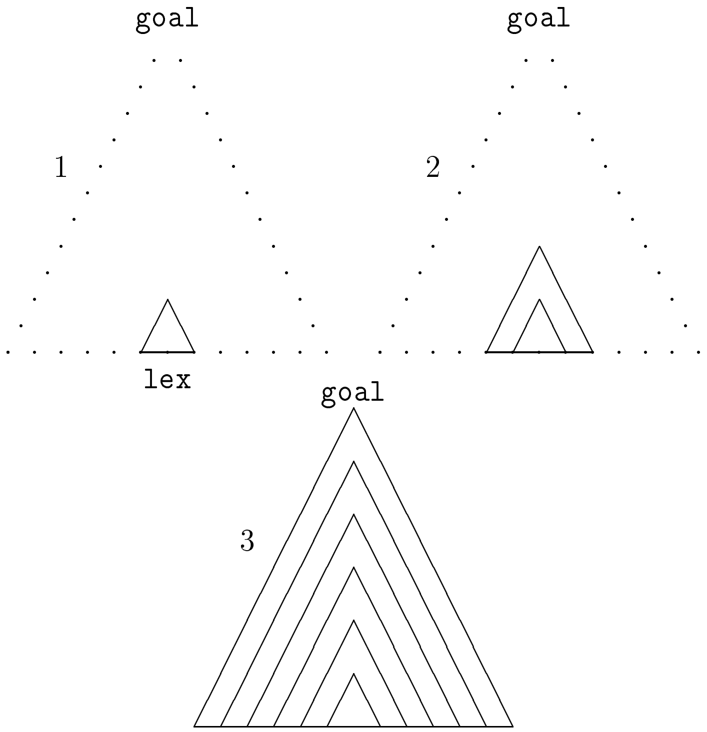


Figure 4.15: The head-corner parser for concatenative grammars. The parser selects the head of the string (1), and proves that this element is the head-corner of the goal. To this end, a rule is selected of which this lexical entry is the head daughter. The other daughters of the rule are parsed recursively. The result is a slightly larger head-corner (2). This process repeats itself until a head-corner is constructed which dominates the whole string (3). The string is recognized bidirectionally.

larger head. This process can be applied iteratively, until the head dominates all words in the string. This is illustrated in figure 4.15. A head-corner chart-parser for context-free grammars is defined in Sikkel and op den Akker (1992). They show that the worst-case complexity of the algorithm is the same as for conventional context-free parsing algorithms such as Earley (1970). The method is comparable to that of Satta and Stock (1989); Satta and Stock (1991).

The head-driven variant of the left-corner algorithm can be adapted to the class of grammars under consideration. Again, the first step of the algorithm consists of the prediction step: which lexical entry is the head corner of the phrase? The first thing to note is that the words introduced by this lexical entry should be part of the input string, because of the non-erasure requirement. Therefore we use the bag of words as a 'guide' (Dymetman *et al.*, 1990) as in a left-corner parser, but we change the way in which lexical entries 'consume the guide'. For the moment I assume that the bag of words is simply represented by the string, but I introduce a better data-structure for bags later in this chapter. The predicates *guide* and *consume_guide* are defined as:

(27) $guide(\text{String}, \text{String}, \langle\rangle)$.

$$
\begin{array}{l}
\textit{consume\_guide}(\langle\rangle, \mathrm{P}, \mathrm{P}). \\
\textit{consume\_guide}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{P}_0, \mathrm{P})\!:\!- \\
\qquad \textit{del}(\mathrm{H}, \mathrm{P}_0, \mathrm{P}_1), \\
\qquad \textit{consume\_guide}(\mathrm{T}, \mathrm{P}_1, \mathrm{P}).
\end{array}
$$

$$
\begin{array}{l}
\textit{del}(\mathrm{El}, \langle \mathrm{El}|\mathrm{T}\rangle, \mathrm{T}). \\
\textit{del}(\mathrm{El}, \langle \mathrm{H}|\mathrm{T}\rangle, \langle \mathrm{H}|\mathrm{T}_2\rangle)\!:\!- \\
\qquad \textit{del}(\mathrm{El}, \mathrm{T}, \mathrm{T}_2).
\end{array}
$$

To instantiate the guide properly, I define the predicate *start_parse*/2 as follows:

(28)
$$
\begin{array}{l}
\textit{start\_parse}(\mathrm{String}, \mathrm{Sign})\!:\!- \\
\qquad \textit{guide}(\mathrm{String}, \mathrm{Guide}, \mathrm{EmptyGuide}) \\
\qquad \textit{parse}(\mathrm{Sign}, \mathrm{Guide}, \mathrm{EmptyGuide}), \\
\qquad \textit{yield}(\mathrm{Sign}, \mathrm{String}).
\end{array}
$$

The interesting predicate is the *parse* predicate. This predicate predicts, for a given goal, its possible head-corner, and then shows that this predicted head-corner, indeed is the head-corner of the goal. As we discussed earlier, in most linguistic theories, it is assumed that certain features are shared between the mother and the head. I assume that the predicate *head*/2 defines these feature percolations; for the grammar of the foregoing section this predicate simply is defined as:

(29) $\textit{head}(\left[\ \textit{syn} : \mathrm{Syn}\ \right], \left[\ \textit{syn} : \mathrm{Syn}\ \right]).$

Because of the definition of 'head corner', these features will also be shared between a node and its head corner; hence we can use this definition to restrict lexical lookup by top-down prediction. [2] The first step in the algorithm is defined as:

(30)
$$
\begin{array}{l}
\textit{parse}(\mathrm{Goal}, \mathrm{P}_0, \mathrm{P})\!:\!- \\
\qquad \textit{predict\_ncr}(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \mathrm{P}_0, \mathrm{P}_1), \\
\qquad \textit{parse\_ds}(\mathrm{Ds}, \mathrm{P}_1, \mathrm{P}_2), \\
\qquad \textit{head\_corner}(\mathrm{Head}, \mathrm{Goal}, \mathrm{P}_2, \mathrm{P}). \\
\textit{parse}(\{\ \mathrm{Call}\ \}, \mathrm{P}, \mathrm{P})\!:\!- \\
\qquad \textit{call}(\mathrm{Call}).
\end{array}
$$

$$
\begin{array}{l}
\textit{parse\_ds}(\langle\rangle, \mathrm{P}, \mathrm{P}). \\
\textit{parse\_ds}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{P}_0, \mathrm{P})\!:\!- \\
\qquad \textit{parse}(\mathrm{H}, \mathrm{P}_0, \mathrm{P}_1), \\
\qquad \textit{parse\_ds}(\mathrm{T}, \mathrm{P}_1, \mathrm{P}).
\end{array}
$$

The relation *parse_ds* simply calls the parse predicate, for each element of a list of daughters. The second clause of the parse predicate is applicable, if one of these daughters is the extra relation call — written between {, }.

The predicates *predict_ncr* predicts for a given goal, a possible head-corner. It is defined as follows:

[2] In the general case we need to compute the transitive closure of (restrictions of (Shieber, 1985)) possible mother-head relationships.

(31) $predict\_ncr(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \mathrm{P}_0, \mathrm{P})$:-
$head(\mathrm{Goal}, \mathrm{Head})$,
$ncr(\mathrm{Head}, \mathrm{Ds}, \mathrm{Words})$,
$consume\_guide(\mathrm{Words}, \mathrm{P}_0, \mathrm{P})$.

Instead of selecting the first word from the current input string, the parser selects a lexical entry dominating a subbag of the words occurring in the input string, provided this lexical entry can be the *head-corner* of the current goal.

The second step of the algorithm, the *head_corner* part, is identical to the *left_corner* part of the left-corner parser, but instead of selecting the leftmost daughter of a rule the head-corner parser selects the head of a rule.

(32) $head\_corner(\mathrm{X}, \mathrm{X}, \mathrm{P}, \mathrm{P})$.
$head\_corner(\mathrm{Small}, \mathrm{Big}, \mathrm{P}_0, \mathrm{P})$:-
$select\_cr(\mathrm{Small}, \mathrm{Mid}, \mathrm{Ds}, \mathrm{P}_0, \mathrm{P}_1)$,
$parse\_ds(\mathrm{Ds}, \mathrm{P}_1, \mathrm{P}_2)$,
$head\_corner(\mathrm{Mid}, \mathrm{Big}, \mathrm{P}_2, \mathrm{P})$.

$select\_cr(\mathrm{Small}, \mathrm{Mid}, \mathrm{Ds}, \mathrm{P}_0, \mathrm{P})$:-
$cr(\mathrm{Small}, \mathrm{Mid}, \mathrm{Ds}, \mathrm{Words})$,
$consume\_guide(\mathrm{Words}, \mathrm{P}_0, \mathrm{P})$.

Example.

To parse the sentence 'dat jan ontwaakt', the head corner parser will proceed as indicated in figure 4.16. Each of the steps will now be clarified as follows.

After the construction of the guide, in the predicate *start_parse*, the first call to *parse* is:

$$\texttt{?-}\quad parse(\begin{bmatrix} syn: \mathrm{comp} \\ sc: \langle\rangle \\ phon: \langle dat, jan, ontwaakt\rangle \end{bmatrix}, \langle dat, jan, ontwaakt\rangle, \langle\rangle).$$

This is the root node in figure 4.16.

The prediction step (1) selects the lexical entry 'dat' (recall that lexical entries are non-chain-rules with an empty list of daughters), because its syntactic features are the same as the syntactic features of this goal, and because the word occurs in the input string. The next goal is to show that this lexical entry is the head corner of the top goal; furthermore the words that still have to be covered are now $\langle jan, ontwaakt\rangle$.

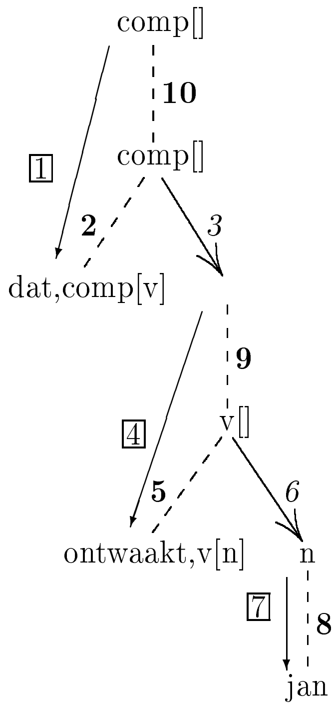


Figure 4.16: Trace of the parse 'dat jan ontwaakt'. The integers associated with the arrows indicate the order of the steps of the parsing process. Prediction steps are indicated with a framed integer, 'head_corner' steps are indicated with a bold face integer, and recursive parse steps are indicated with a slanted integer. The nodes represent (some of) the information available at the moment this step is performed. The left-most daughter of each local tree corresponds to the head daughter.

The *head_corner* clause looks, essentially, as follows (step 2):

$$
\texttt{?-}\quad \mathit{head\_corner}(\left[\begin{array}{l}
\mathit{syn} : \mathrm{comp} \\
\mathit{sc} : \langle \left[\begin{array}{l} \mathit{syn} : \mathrm{v} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{Sem} \\ \mathit{rule} : \mathrm{right} \end{array}\right] \rangle \\
\mathit{h\text{-}s} : \left[\begin{array}{l} \mathit{left} : \langle\rangle \\ \mathit{head} : \langle \mathrm{dat} \rangle \\ \mathit{right} : \langle\rangle \end{array}\right] \\
\mathit{phon} : \langle \mathrm{dat} \rangle \\
\mathit{sem} : \mathrm{Sem}
\end{array}\right],
\left[\begin{array}{l}
\mathit{syn} : \mathrm{comp} \\
\mathit{sc} : \langle\rangle \\
\mathit{phon} : \langle \mathrm{dat, jan, ontwaakt} \rangle
\end{array}\right],
\langle \mathrm{jan, ontwaakt} \rangle, \langle\rangle).
$$

The hypothesized head-corner, the lexical entry 'dat', has to be matched with the head of a rule. Notice that 'dat' subcategorizes for a sign with syntactic category V, hence the next goal (3) is to parse the V; the guide is instantiated as the bag 'jan, ontwaakt':

$$
\texttt{?-}\quad \mathit{parse}(\left[\begin{array}{l}
\mathit{syn} : \mathrm{v} \\
\mathit{sc} : \langle\rangle \\
\mathit{sem} : \mathrm{Sem} \\
\mathit{rule} : \mathrm{right}
\end{array}\right], \langle \mathit{jan}, \mathit{ontwaakt} \rangle, \mathrm{P}).
$$

For this goal, the prediction step selects the word *ontwaakt* (4). Next, the word *ontwaakt* has to be shown to be the head corner of the V goal, by the *head_corner* predicate (5):

$$
\texttt{?-}\quad \mathit{head\_corner}(\left[\begin{array}{l}
\mathit{syn} : \mathrm{v} \\
\mathit{sc} : \langle \left[\begin{array}{l} \mathit{syn} : \mathrm{n} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{Subj} \\ \mathit{rule} : \mathrm{left} \end{array}\right] \rangle \\
\mathit{h\text{-}s} : \left[\begin{array}{l} \mathit{left} : \langle\rangle \\ \mathit{head} : \langle \mathrm{ontwaakt} \rangle \\ \mathit{right} : \langle\rangle \end{array}\right] \\
\mathit{sem} : \mathrm{ontwaakt(Subj)} \\
\mathit{phon} : \langle \mathrm{ontwaakt} \rangle
\end{array}\right],
\left[\begin{array}{l}
\mathit{syn} : \mathrm{v} \\
\mathit{sc} : \langle\rangle \\
\mathit{sem} : \mathrm{Sem} \\
\mathit{rule} : \mathrm{right}
\end{array}\right], \langle \mathrm{jan} \rangle, \mathrm{P}).
$$

In order to prove this predicate, the next parse goal consists in parsing a NP (for which *ontwaakt* subcategorizes, step 6); the guide is the bag consisting of the element *jan*. This goal succeeds: predict a possible head corner (7), the lexical entry ‘jan’, which is a trivial head corner of the NP (8). After the successful parse of the NP, this NP is combined with the V, with the *cp* predicate. Note that the verb selected an NP with rule-feature ‘left’, and hence the phonology of the NP is appended to the left of ‘ontwaakt’. As the verb does not select any other elements, the resulting verb phrase is indeed a possible head-corner of the parse goal above (9), instantiating this goal to:

$$\texttt{?-}\quad \mathit{parse}(\begin{bmatrix} \mathit{syn} : \mathrm{v} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{ontwaakt(jan)} \\ \mathit{h\text{-}s} : \begin{bmatrix} \mathit{left} : \langle \mathrm{jan} \rangle \\ \mathit{head} : \langle \mathrm{ontwaakt} \rangle \\ \mathit{right} : \langle\rangle \end{bmatrix} \\ \mathit{phon} : \langle \mathrm{jan}, \mathrm{ontwaakt} \rangle \\ \mathit{rule} : \mathrm{right} \end{bmatrix}, \langle \mathrm{jan}, \mathrm{ontwaakt} \rangle, \langle\rangle).$$

and therefore we now have found the V for which *dat* subcategorizes. The next task is to combine this verb phrase with the lexical entry ‘dat’, by means of the ‘cp’ predicate. The verb phrase is selected by ‘dat’, with rule feature ‘right’. Therefore the phonology of the verb phrase is appended to the right of the phonology of this complementizer. The next goal (10) is to connect the complementizer with an empty subcat list up to the top-goal, with trivial success. Hence the first call to parse will succeed, yielding:

$$\texttt{?-}\quad \mathit{parse}(\begin{bmatrix} \mathit{syn} : \mathrm{comp} \\ \mathit{sc} : \langle\rangle \\ \mathit{sem} : \mathrm{dat(ontwaakt(jan))} \\ \mathit{h\text{-}s} : \begin{bmatrix} \mathit{left} : \langle\rangle \\ \mathit{head} : \langle \mathrm{dat} \rangle \\ \mathit{right} : \langle \mathrm{jan}, \mathrm{ontwaakt} \rangle \end{bmatrix} \\ \mathit{phon} : \langle \mathrm{dat}, \mathrm{jan}, \mathrm{ontwaakt} \rangle \end{bmatrix}, \langle \mathrm{dat}, \mathrm{jan}, \mathrm{ontwaakt} \rangle, \langle\rangle).$$

A slightly more complex example is shown in figure 4.17, for the sentence

(33) Hoort Arie Bob sla bestellen?
Hears Arie Bob salad order
Does Arie hear that Bob orders salad?

4.5 Head-driven parsing for TAGs

This section shows how the head-corner parser might be applicable for lexicalized and constraint-based versions of TAGs. An auxiliary tree will correspond to a *headed* rule, i.e. *chain-rule*. Initial trees, on the other hand, will correspond to *non-chain-rules*.

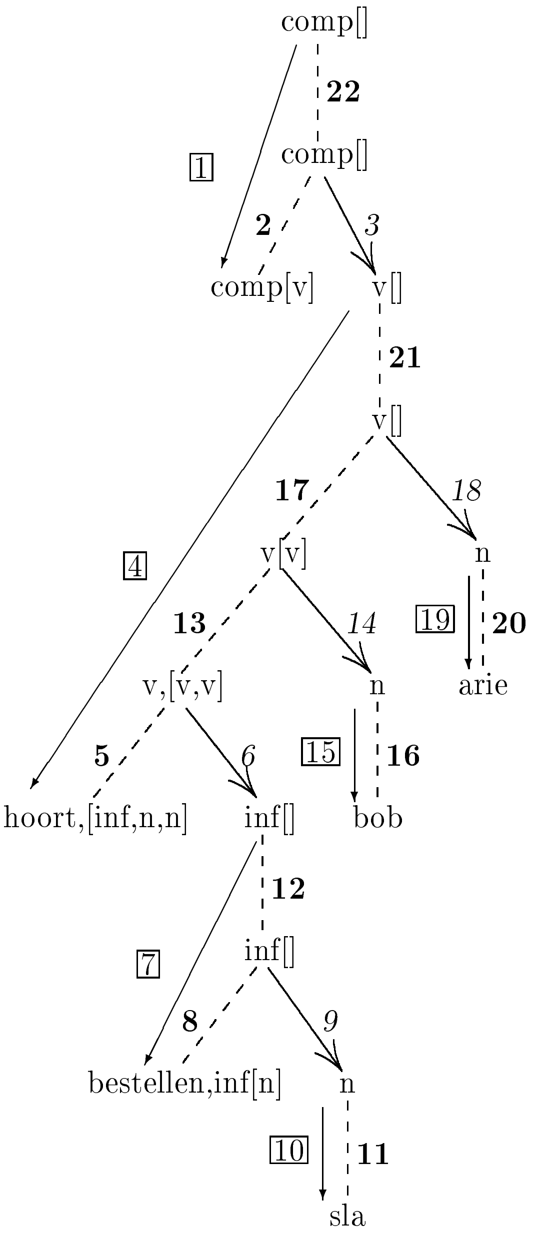


Figure 4.17: Parsing ‘Hoort Arie Bob sla bestellen’

Firstly, I show how initial and auxiliary trees are encoded in $\mathcal{R}(\mathcal{L})$. Given these data-structures, I define the adjunction operation. Furthermore I discuss how the unification of bottom and top feature structures comes about at the end of a derivation. In order to prevent spurious ambiguities I then define 'normal form' derivations. This allows the parser to implement the unification of bottom and top feature structures in an incremental manner.

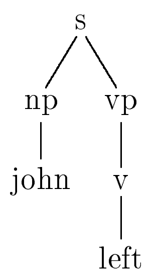
An important reduction of the search space is possible, because it is not possible during a TAG derivation to change the order of words, once this order has been established. Thus, TAG derivations exhibit a certain monotonicity with respect to the order of the words. Therefore, an important reduction of the search space is obtained by checking at various moments during the parse whether the structure obtained yields a subsequence of the string to be parsed.

Although it is possible to present the head-corner parser for TAG simply as an instantiation of the algorithm presented in the previous section, it is somewhat more easier to understand the parser, if we change various parts of the head-corner parser directly, to exhibit more clearly what is going on.

4.5.1 Representing auxiliary and initial trees

In order to use the head-driven parser for TAG I describe how trees are encoded in $\mathcal{R}(\mathcal{L})$, and how a given TAG is represented by a set of definite clauses. I then describe how this set of clauses is divided in chain- and non-chain-rules.

Trees are represented by feature structures with attributes *node*, *mrk*, *ds* and *term* where the value of *node* represents the node label, *mrk* is a marker of which the role will be explained below, and *ds* is a list of daughter trees (in case of non-terminal nodes) or a list of words (in case of terminal nodes). The attribute *term* takes values yes, no depending on whether the node is terminal or not. Without loss of generality I furthermore assume that node labels consist of a feature structure with three attributes *cat*, *bot* and *top* of which the values are resp. the category label, the bottom features and the top features. Thus, the following tree:



is represented as a feature structure as follows (note that in this example no constraints are defined, hence the values of the attributes *bot* and *top* are unspecified)

$$\begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{s} \end{bmatrix} \\ term: \mathrm{no} \\ ds: \langle \begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{np} \end{bmatrix} \\ term: \mathrm{yes} \\ ds: \langle \mathrm{john} \rangle \end{bmatrix}, \begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{vp} \end{bmatrix} \\ term: \mathrm{no} \\ ds: \langle \begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{v} \end{bmatrix} \\ term: \mathrm{yes} \\ ds: \langle \mathrm{left} \rangle \end{bmatrix} \rangle \end{bmatrix} \rangle \end{bmatrix}$$

This data structure is chosen to implement the idea that at least some information between bottom and top parts of labels is shared; this part is represented as the value of the *cat* attribute. This is useful to provide an appropriate definition for the 'head' relation for TAG. This definition reads

(34) $head(\begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{C} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} node: \begin{bmatrix} cat: \mathrm{C} \end{bmatrix} \end{bmatrix}).$

If we were to write a TAG grammar, as a grammar of $\mathcal{R}(\mathcal{L})$, then an initial tree Tree with substitution nodes $\mathrm{D}_1 \ldots \mathrm{D}_n$ were to be defined as:

$sign(\mathrm{Tree})\,{:}{-}\,sign(\mathrm{D}_1), \ldots sign(\mathrm{D}_n), \phi.$

An auxiliary tree Tree, with foot node Foot and substitution nodes $\mathrm{D}_1 \ldots \mathrm{D}_n$, on the other hand, were to be written as:

$sign(\mathrm{M})\,{:}{-}\,sign(\mathrm{H}), sign(\mathrm{D}_1) \ldots sign(\mathrm{D}_n), adjoin(\mathrm{Tree}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{M}), \phi.$

However, for the meta-interpreter, we will not assume this representation, but assume that an initial tree is represented as a clause

(35) $init(\mathrm{Tree}, \mathrm{SubsNodes}, \mathrm{Words})\,{:}{-}\,\phi.$

where Tree is the partially instantiated initial tree; SubsNodes is a list of the uninstantiated substitution nodes in Tree, and Words are the words dominated by this initial tree. An auxiliary tree Aux with foot node Foot, substitution nodes SubsNodes and words Words is represented as

(36) $aux(\mathrm{Tree}, \mathrm{Foot}, \mathrm{SubsNodes}, \mathrm{Words})\,{:}{-}\,\phi.$

The basic idea will be that an initial tree corresponds to a non-chain-rule. The 'daughters' of such a rule correspond to the substitution nodes of that initial tree. An auxiliary tree, on the other hand, corresponds to a chain-rule. The head of the rule will be the tree in which the auxiliary is adjoined; the mother of the rule will be the result of the adjunction. The other daughters of the rule will correspond to the substitution nodes of the auxiliary tree. The important part of this definition is the introduction, at the end of the list of daughters, of the relation 'adjoin' which will be explained below. The following definitions of the predicates *predict_ncr* and *select_cr* are obtained. Note that the use of the fourth argument of the latter predicate will become clear later.

(37) $\textit{predict\_ncr}(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \mathrm{P}_0, \mathrm{P})$:-
$\quad\quad \textit{head}(\mathrm{Goal}, \mathrm{Head}),$
$\quad\quad \textit{init}(\mathrm{Head}, \mathrm{Ds}, \mathrm{Words}),$
$\quad\quad \textit{consume\_guide}(\mathrm{Words}, \mathrm{P}_0, \mathrm{P}).$

$\quad \textit{select\_cr}(\mathrm{Small}, \mathrm{Mid}, \mathrm{Ds}, \mathrm{Foot}, \mathrm{P}_0, \mathrm{P})$:-
$\quad\quad \textit{aux}(\mathrm{Aux}, \mathrm{Foot}, \mathrm{Subs}, \mathrm{Words}),$
$\quad\quad \textit{consume\_guide}(\mathrm{Words}, \mathrm{P}_0, \mathrm{P}_1),$
$\quad\quad \textit{adjoin}(\mathrm{Small}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{Mid}).$

4.5.2 Adjunction.

The adjunction relation is defined as a relation between an (input) tree Tree, an auxiliary tree Aux of which the foot node is Foot and an (output) tree NewTree. The relation is non-deterministic with respect to which node in Tree the auxiliary tree is adjoined. Therefore, the first clause of *adjoin* states that the relation is true if the auxiliary tree is adjoined in one of the daughters of the input tree; the second clause states that the relation is true if the auxiliary tree is adjoined at the root node of the input tree:

$$(38)\quad \textit{adjoin}(\begin{bmatrix} \textit{node} : \textit{cat} : \mathrm{Cat} \\ \textit{mrk} : \mathrm{none} \\ \textit{term} : \mathrm{no} \\ \textit{ds} : \mathrm{Ds} \end{bmatrix}, \mathrm{Aux}, \mathrm{Foot}, \begin{bmatrix} \textit{node} : \textit{cat} : \mathrm{Cat} \\ \textit{mrk} : \mathrm{none} \\ \textit{term} : \mathrm{no} \\ \textit{ds} : \mathrm{Ds}_2 \end{bmatrix}) :-$$

$\quad\quad \textit{adjoin\_ds}(\mathrm{Ds}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{Ds}_2).$
$\quad \textit{adjoin}(\mathrm{In}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{Aux})$:-
$\quad\quad \textit{adj\_now}(\mathrm{In}, \mathrm{Aux}, \mathrm{Foot}).$

For the moment, the reader should not worry about the *mrk* attribute. Its purpose will be explained later. The predicate *adjoin_ds* non-deterministically chooses a daughter from the list of daughters to adjoin the auxiliary tree in:

(39) $\textit{adjoin\_ds}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{Aux}, \mathrm{Foot}, \langle \mathrm{H}_2|\mathrm{T}\rangle)$:-
$\quad\quad \textit{adjoin}(\mathrm{H}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{H}_2).$
$\quad \textit{adjoin\_ds}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{Aux}, \mathrm{Foot}, \langle \mathrm{H}|\mathrm{T}_2\rangle)$:-
$\quad\quad \textit{adjoin\_ds}(\mathrm{T}, \mathrm{Aux}, \mathrm{Foot}, \mathrm{T}_2).$

The second clause for *adjoin* states that adjunction may take place at the root node of the input tree. The resulting tree is defined as the auxiliary tree of which the foot node is instantiated with the daughters of the input tree. Furthermore we take care of the unification of the bottom and top features. The top features of the root node of the input tree are unified with the top features of the root node of the auxiliary tree; the bottom features of the root node of the input tree are unified with the bottom features of the foot node (recall figure 4.10).

$$(40)\ \ adj\_now(\begin{bmatrix} node : \begin{bmatrix} cat : \mathrm{C} \\ top : \mathrm{T} \\ bot : \mathrm{B} \end{bmatrix} \\ mrk : \mathrm{none} \\ ds : \mathrm{Ds} \\ term : \mathrm{Term} \end{bmatrix}, \begin{bmatrix} node : \begin{bmatrix} cat : \mathrm{C} \\ top : \mathrm{T} \end{bmatrix} \end{bmatrix}, \begin{bmatrix} node : \begin{bmatrix} cat : \mathrm{C} \\ bot : \mathrm{B} \end{bmatrix} \\ mrk : \mathrm{none} \\ ds : \mathrm{Ds} \\ term : \mathrm{Term} \end{bmatrix}).$$

4.5.3 String concatenation

In order to check what string a given derived tree dominates, we could define the relation 'yield' which – procedurally speaking – travels a tree in a top-down fashion and collects the terminal symbols at the leaves of the tree. However, note that TAG derivations exhibit a monotonicity property with respect to the order of words. Once a certain order has been established, this order cannot be changed anymore. For this reason, all trees which are derived during a derivation yield a string which is a subsequence of the string to be parsed. For that reason an important reduction of the search space can be obtained by checking whether an hypothesized derived tree indeed yields a subsequence of the string to be parsed. Furthermore, such a check then also implies that it is not necessary anymore, to check whether the topmost derived tree yields the desired string: this is necessarily the case, because a topmost derived tree has used all words in the input and furthermore yields a subsequence.

The following predicates are obtained. Note that we now assume an extra argument position through which we percolate the input string. The predicate *head_corner* will be modified later.

(41) *start_parse*(String, Sign) :-
    guide(String, Guide, EmptyGuide),
    parse(Sign, String, Guide, EmptyGuide).

parse(Goal, Str, P_0, P) :-
    predict_ncr(Goal, Head, Ds, P_0, P_1),
    parse_ds(Ds, P_1, P_2),
    yield_subseq(Head, Str),
    head_corner(Head, Goal, Str, P_2, P).

parse_ds($\langle\rangle$, Str, P, P).
parse_ds($\langle$H|T$\rangle$, Str, P_0, P) :-
    parse(H, Str, P_0, P_1),
    parse_ds(T, Str, P_1, P).

(42) `% head_corner is modified later.`
head_corner(X, X, Str, P, P).
head_corner(Small, Big, Str, P_0, P) :-
    select_cr(Small, Mid, Subs, Foot, P_0, P_1),
    parse_ds(Subs, P_1, P_2),

$$\begin{array}{l} \quad\quad \textit{yield\_subseq}(\mathrm{Mid}, \mathrm{Str}), \\ \quad\quad \textit{head\_corner}(\mathrm{Mid}, \mathrm{Big}, \mathrm{Str}, \mathrm{P}_2, \mathrm{P}). \end{array}$$

The predicate *yield_subseq* straightforwardly checks whether the first argument, a tree, yields a subsequence of the second argument, a string.

(43) $$\begin{array}{l} \textit{yield\_subseq}(\begin{bmatrix} \textit{term} : \mathrm{T}, \\ \textit{ds} : \mathrm{Ds} \end{bmatrix}, \mathrm{Str})\texttt{:-} \\ \quad\quad \textit{yield\_subseq}(\mathrm{T}, \mathrm{Ds}, \mathrm{Str}, \_). \end{array}$$

$$\begin{array}{l} \textit{yield\_subseq}(\mathrm{no}, \langle\rangle, \mathrm{S}, \mathrm{S}). \\ \textit{yield\_subseq}(\mathrm{no}, \langle \begin{bmatrix} \textit{term} : \mathrm{Term} \\ \textit{ds} : \mathrm{Ds} \end{bmatrix} | \mathrm{T}\rangle, \mathrm{S}_0, \mathrm{S})\texttt{:-} \\ \quad\quad \textit{yield\_subseq}(\mathrm{Term}, \mathrm{Ds}, \mathrm{S}_0, \mathrm{S}_1), \\ \quad\quad \textit{yield\_subseq}(\mathrm{no}, \mathrm{T}, \mathrm{S}_1, \mathrm{S}). \\ \textit{yield\_subseq}(\mathrm{yes}, \mathrm{W}, \mathrm{S}_0, \mathrm{S})\texttt{:-} \\ \quad\quad \textit{subsequence}(\mathrm{W}, \mathrm{S}_0, \mathrm{S}). \end{array}$$

$$\begin{array}{l} \textit{subsequence}(\langle\rangle, \mathrm{S}, \mathrm{S}). \\ \textit{subsequence}(\langle \mathrm{H}|\mathrm{T}\rangle, \langle \mathrm{H}|\mathrm{T}_2\rangle, \mathrm{S})\texttt{:-} \\ \quad\quad \textit{subsequence}(\mathrm{T}, \mathrm{T}_2, \mathrm{S}) \\ \textit{subsequence}(\langle \mathrm{H}|\mathrm{T}\rangle, \langle \_|\mathrm{T}_2\rangle, \mathrm{S})\texttt{:-} \\ \quad\quad \textit{subsequence}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{T}_2, \mathrm{S}) \end{array}$$

4.5.4 Spurious ambiguity

The way in which adjunction and substitution is defined in previous subsections give rise to spurious ambiguities. For example, given two auxiliary trees β_5 and β_6 and a derived tree γ, the parser derives the application of β_5 and β_6 to γ twice, corresponding to the applications $\beta_5(\beta_6(\gamma))$ and $\beta_6(\beta_5(\gamma))$. But the results of these derivations are clearly completely identical.

As an example, consider the following case in which γ, β_5 and β_6 are as in figure 4.18.

The application $\beta_5(\beta_6(\gamma))$ gives rise to two possible derived trees, depending on which node with category `n`, is chosen for adjunction (see figure 4.19 for an illustration). Clearly, the application $\beta_6(\beta_5(\gamma))$ gives the same two results. The parser thus delivers four results, where only two results are desirable.

Another spurious ambiguity arises when a tree is substituted in another tree, and afterwards adjunction in this tree is possible. The parser will prove two alternative derivations. The first derivation corresponds to the case in which the auxiliary tree is first adjoined in the derived tree, which then is substituted in the main tree. The second derivation corresponds to the case where the derived tree is substituted in the main tree; the auxiliary tree is afterwards adjoined (deep down) in this main tree.

It is straightforward to solve these problems by the following two principles. Firstly, once a tree is substituted in another tree this sub-tree is regarded 'completed'. This

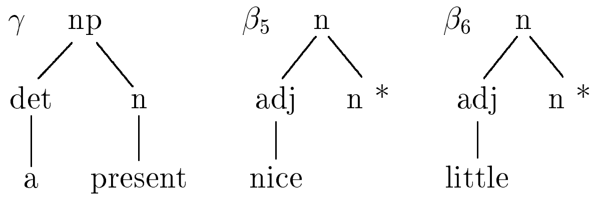


Figure 4.18: A derived tree and two auxiliary trees, which are used to illustrate the problem of spurious ambiguity.

means that no adjunctions take place in this tree. Similarly, if an auxiliary tree is adjoined at some node in a derived tree, then the sub-tree dominated by the foot node (in the result of the adjunction) is 'completed' as well, and no further adjunctions under this foot node are possible. In the illustration I mark such 'completed' sub-trees with the '#' marker.

The example given above is now derived only twice as follows. Firstly, the application $\beta_6(\gamma)$ gives rise to a derived tree in which the embedded n node is marked completed. For that reason applying β_5 to it only gives rise to one possibility, because the marker on the node $\mathtt{n}_1$ prevents adjunction at $\mathtt{n}_1$, and hence only adjunction at node $\mathtt{n}_2$ is possible (see figure 4.20 for illustration). The derivation $\beta_6(\beta_5(\gamma))$ produces the other possibility.

In this way, trees are constructed in a bottom up fashion. Once you adjoin at a given node, then everything under (the bottom part) of this node is completed.

The prevention of spurious ambiguity is easily implemented using the attribute *mrk* in the data structure representing trees. The node in a tree representing a substitution node will be marked 'completed'. Note that this marker remains after substitution. Similarly, the marker of a node representing a foot node will be marked 'completed'. This changes the representation of initial and auxiliary trees. The adjunction predicate, furthermore, is defined in such a way that it may only penetrate nodes, which are not marked 'completed'. Note that in the definition of *adjoin* this is already achieved.

4.5.5 Unification of bottom and top.

At the end of a derivation the bottom- and top-parts of a node are to be unified. The relation *unify_nodes* is true of trees of which the nodes have identical bottom- and top values. Note that this predicate essentially implements a non-monotonic device, and hence cannot be straightforwardly defined in a grammar of $\mathcal{R}(\mathcal{L})$; on the other hand, it is quite easy to augment the meta-interpreter to implement this device, as part of the definition of *start_parse*. Thus, to ensure that the nodes of the trees built by the parser have identical bottom- and top parts, the relation 'unify_nodes' could be added to the definition of *start_parse*.

However, once it is known that at some node, no further adjunctions are possible,

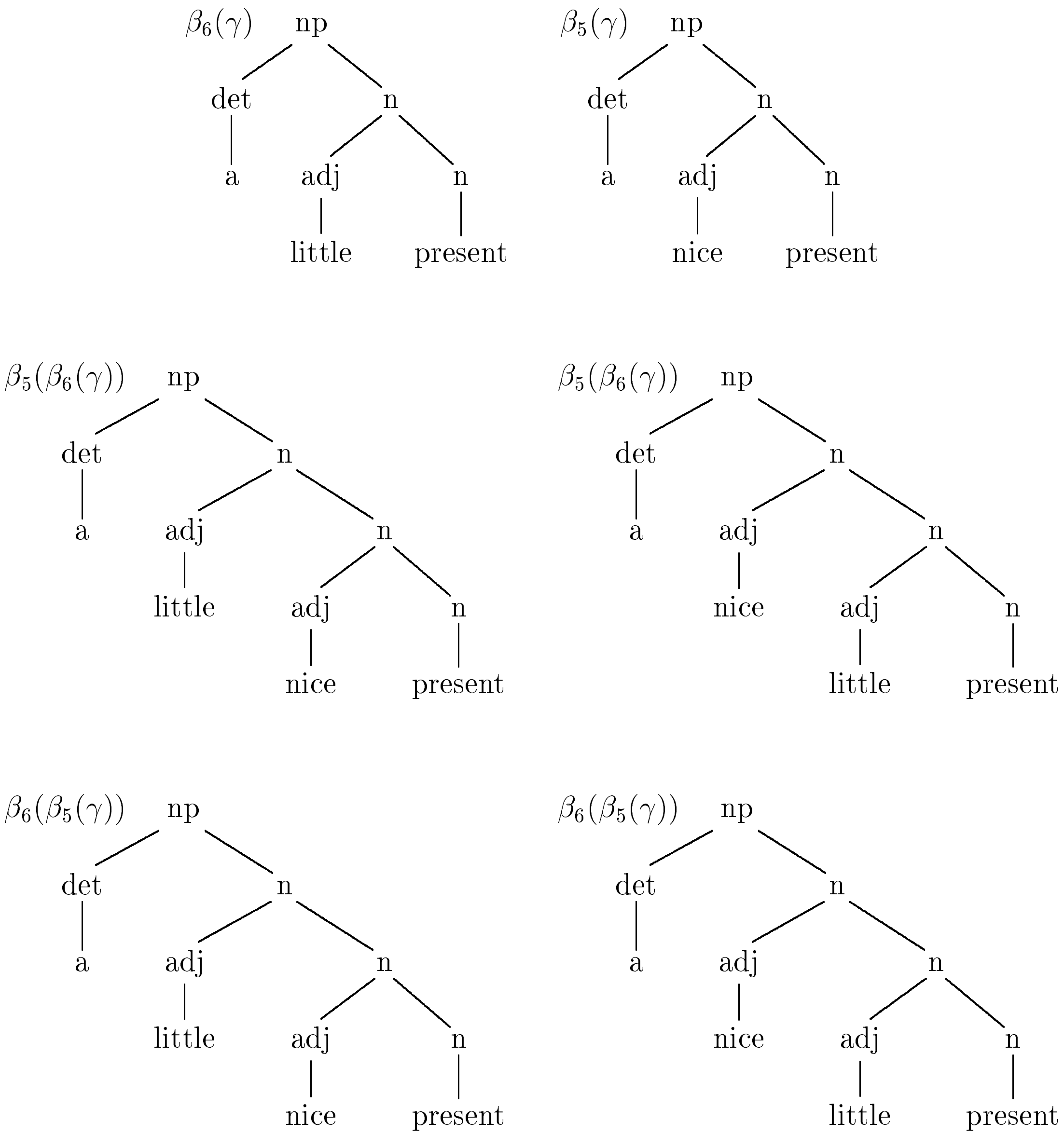


Figure 4.19: Illustration of spurious ambiguities. The first two trees respectively corresponds to $\beta_6(\gamma)$ and $\beta_5(\gamma)$. Depending on the adjunction node, these two trees both can be expanded in two ways, by the adjunction of the other auxiliary tree, resulting in two trees corresponding to $\beta_5(\beta_6(\gamma))$, and two trees corresponding to $\beta_6(\beta_5(\gamma))$. Four results are derived by the parser, where only two should be derived.

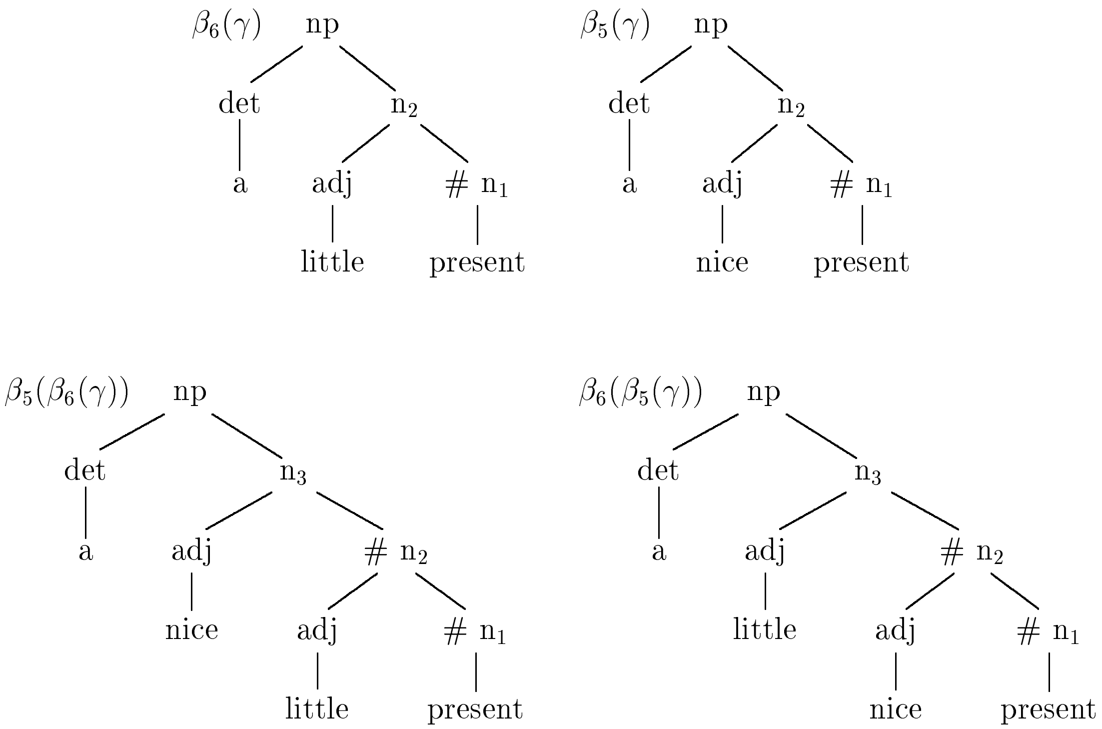


Figure 4.20: Preventing spurious ambiguity. Because of the marking of nodes in the subtree of a foot node, no spurious ambiguity arises.

then we might just as well unify the bottom- and top parts of that node immediately. This may be useful, in cases where this unification fails — in that case we may abandon a search path without a solution much earlier. Thus, it is possible to implement the *unify_nodes* predicate in an incremental fashion.

The incremental unification of top and bottom features is easily implemented. Firstly, the predicate *unify_nodes* is defined in such a way that completed nodes are not penetrated: these nodes are already unified in a previous cycle. Furthermore, this predicate is now called when partial trees are completed; that is, after a substitution, and after an adjunction.

The predicate *unify_nodes* is defined as follows.

(44) $\mathit{unify\_nodes}(\left[\ \mathit{mrk} : \text{completed}\ \right]).$

$$\mathit{unify\_nodes}(\left[\begin{array}{l} \mathit{node} : \left[\begin{array}{l} \mathit{bot} : \mathrm{X} \\ \mathit{top} : \mathrm{X} \end{array}\right] \\ \mathit{mrk} : \text{none} \\ \mathit{term} : \mathrm{T} \\ \mathit{ds} : \mathrm{Ds} \end{array}\right]) \texttt{:-}$$

$\qquad \mathit{unify\_ds}(\mathrm{T}, \mathrm{Ds}).$

$\mathit{unify\_ds}(\text{yes}, \_).$
$\mathit{unify\_ds}(\text{no}, \langle\rangle).$
$\mathit{unify\_ds}(\text{no}, \langle \mathrm{H}|\mathrm{T}\rangle) \texttt{:-}$
$\qquad \mathit{unify\_nodes}(\mathrm{H}),$
$\qquad \mathit{unify\_ds}(\text{no}, \mathrm{T}).$

The predicate *head_corner* is changed as follows:

(45) $\mathit{head\_corner}(\mathrm{X}, \mathrm{X}, \mathrm{Str}, \mathrm{P}, \mathrm{P}) \texttt{:-}$
$\qquad \mathit{unify\_nodes}(\mathrm{X}).$
$\mathit{head\_corner}(\mathrm{Small}, \mathrm{Big}, \mathrm{Str}, \mathrm{P}_0, \mathrm{P}) \texttt{:-}$
$\qquad \mathit{select\_cr}(\mathrm{Small}, \mathrm{Mid}, \mathrm{Subs}, \mathrm{Foot}, \mathrm{P}_0, \mathrm{P}_1),$
$\qquad \mathit{parse\_ds}(\mathrm{Subs}, \mathrm{P}_1, \mathrm{P}_2),$
$\qquad \mathit{yield\_subseq}(\mathrm{Mid}, \mathrm{Str}),$
$\qquad \mathit{unify\_nodes}(\mathrm{Foot}),$
$\qquad \mathit{head\_corner}(\mathrm{Mid}, \mathrm{Big}, \mathrm{Str}, \mathrm{P}_2, \mathrm{P}).$

4.5.6 Examples

Consider the following TAG in figure 4.21.

As an example of the encoding of such elementary trees in $\mathcal{R}(\mathcal{L})$, consider the encoding of α_1 in figure 4.22 and the encoding of β_3 in figure 4.23.

Consider what happens if we parse the sentence "the very pretty girl left today". The first goal of the parser, is to find a tree of which the root node has category `s`. In order to find such a goal, a non-chain-rule is selected which has also `s` as its root node, and of which the string is part of the input string. The rule which is selected is rule

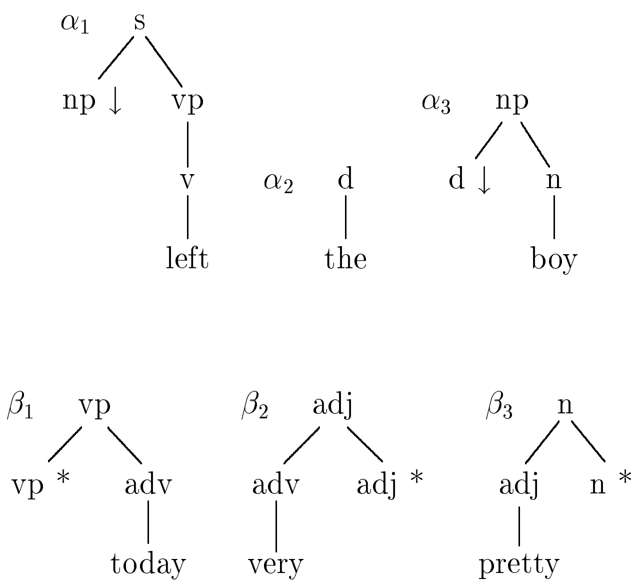
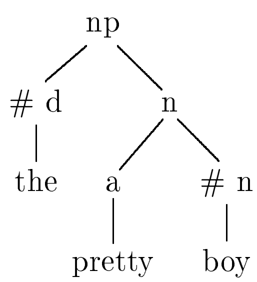


Figure 4.21: Initial and auxiliary trees of example TAG.

α_1. The daughter of this rule is the substitution node `np`. Therefore, the embedded parse goal is to parse an `np`, with bag of words *[the,very,pretty,girl,today]*. Again, the first step of the parser consists in the prediction of a non-chain-rule, of which the root has category `np`, and of which the string is part of the input bag. This time α_2 is selected and we obtain another embedded parse goal: the parsing of a `d` with bag *[the,very,pretty,today]*. The non-chain-rule which is selected for this goal is α_3. As this tree does not have any substitution nodes, we can immediately connect α_3 to the goal `d`. As no auxiliary trees apply to α_3, connection is trivial, and we finish the embedded parse goal for `d`. We thus continue parsing of an `np`, with head α_2, and of which the substitution nodes are filled in, in the mean time. To connect this tree upward to the `np` goal, the auxiliary rule β_3 may be applied. After the application of that auxiliary tree we obtain the tree:



Again, this tree should be connected upward to the NP goal, with input bag *[very,today]*,

$$
\text{(46)}\quad \mathit{init}(\mathrm{M}, \langle \mathrm{D}_1 \rangle, \langle \mathit{left} \rangle)\,\texttt{:-}
$$

$$
\left[\begin{array}{l}
\mathit{mrk} : \text{none} \\
\mathit{node} : \mathit{cat} : \text{s} \\
\mathit{term} : \text{no} \\
\mathit{ds} : \left\langle \begin{array}{l}
\left[\begin{array}{l}
\mathit{mrk} : \text{completed} \\
\mathit{node} : \mathrm{N} \\
\mathit{term} : \mathrm{T} \\
\mathit{ds} : \mathrm{D}
\end{array}\right], \\
\left[\begin{array}{l}
\mathit{mrk} : \text{none} \\
\mathit{node} : \mathit{cat} : \text{vp} \\
\mathit{term} : \text{no} \\
\mathit{ds} : \left\langle \left[\begin{array}{l}
\mathit{mrk} : \text{none} \\
\mathit{node} : \mathit{cat} : \text{v} \\
\mathit{term} : \text{yes} \\
\mathit{ds} : \langle \text{left} \rangle
\end{array}\right] \right\rangle
\end{array}\right]
\end{array} \right\rangle
\end{array}\right]_{\mathrm{M}} ,
$$

$$
\left[\begin{array}{l}
\mathit{mrk} : \text{none} \\
\mathit{node} : \left[\ \mathit{cat} : \text{np}\ \right]_N \\
\mathit{term} : \mathrm{T} \\
\mathit{ds} : \mathrm{D}
\end{array}\right]_{\mathrm{D}_1} .
$$

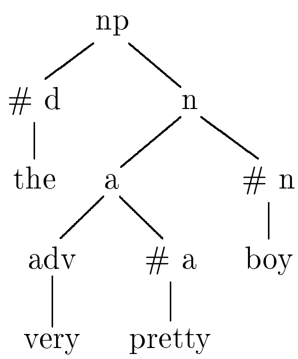
Figure 4.22: Encoding of α_1 as a unit clause of $\mathcal{R}(\mathcal{L})$.

$$
(47)\ \ aux(\mathrm{Aux}, \mathrm{Foot}, \langle\rangle, \langle pretty \rangle)\,\text{:-}
$$

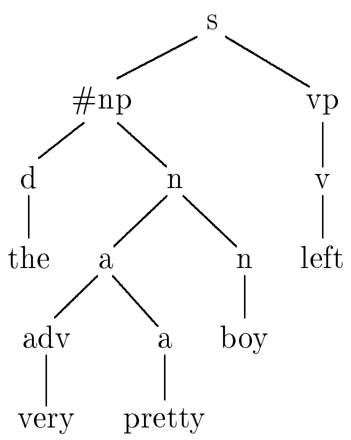
$$
\left[\begin{array}{l}
mrk : \text{none} \\
node : cat : \text{n} \\
term : \text{no} \\
ds : \left\langle \begin{array}{l}
\left[\begin{array}{l} mrk : \text{none} \\ node : cat : \text{adj} \\ term : \text{yes} \\ ds : \langle \text{pretty} \rangle \end{array}\right], \\
\left[\begin{array}{l} mrk : \text{completed} \\ node : \left[\, cat : \text{n} \,\right]_{\text{F}} \\ term : \text{T} \\ ds : \text{D} \end{array}\right]
\end{array} \right\rangle
\end{array}\right]_{\text{Aux}},
\left[\begin{array}{l}
mrk : \text{none} \\
node : \text{F} \\
term : \text{T} \\
ds : \text{D}
\end{array}\right]_{\text{Foot}}.
$$

Figure 4.23: Encoding of β_3 as a unit clause of $\mathcal{R}(\mathcal{L})$.

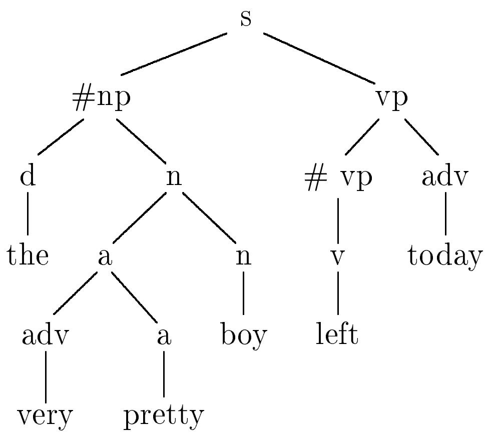
and another auxiliary tree can be applied: α_2, giving tree:



Finally, this connects the input tree to the `np` goal, and this embedded parse is finished. Therefore, we continue the parsing of the `s` goal, with input bag *[today]*, and of which the head is instantiated as:



The auxiliary rule β_1 applies, giving the tree:



which we connect trivially to the goal, as there are no more words left in the input bag.

This example clearly shows how the parser first selects heads, then parses arguments, and finally parses adjunctions.

4.5.7 Semi-lexicalized TAG

The proposed head-driven algorithm for TAG terminates for all lexicalized TAGs. Note though that the algorithm terminates for a strictly larger set of TAGs. Consider the class of TAGs in which each initial tree is lexical, and where furthermore each auxiliary tree is branching. That is, we do not require that auxiliary trees have an anchor, as long as they have (at least) one substitution node. This for example allows an auxiliary tree like the following in figure 4.24: We might call grammars that allow such auxiliary trees 'semi-lexicalized'.

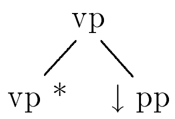


Figure 4.24: Example of auxiliary tree in semi-lexicalized TAG. A Tag is semi-lexicalized in case its initial trees are all lexicalized, and its auxiliary trees are either lexicalized or branching.

Such auxiliary trees may in fact be very useful. To treat modification with auxiliary trees in a lexicalized TAG we need to assume, for example, that each preposition is ambiguous, given that prepositional phrases may modify different categories (at least noun phrases, verb phrases and adjectival phrases), and may occur as arguments. In semi-lexicalized TAGs on the other hand we can simply assume that a preposition corresponds to an initial tree. Furthermore for each possible modification there is one auxiliary tree.

It is not difficult to see that the head-driven parser in fact terminates for semi-lexicalized grammars. This is so, because adjunctions are seen as chain-rules and hence applied in a bottom-up fashion. For that reason no left-recursion can arise: the trees that are derived always grow (in terms of the length of their yield).

4.6 Discussion and Extensions

The algorithm as it is defined is *sound* and *complete* in the usual depth-first, backtrack search sense. Clearly the parser may enter an infinite loop (in case non branching rules are defined that may feed themselves or in case a grammar makes a heavy use of empty categories). However, in case the parser *does* terminate one can be sure that it has found all solutions. The parser always terminates, in case the grammar solely consists of non-chain-rules which consumes some input, and chain-rules which either branch or consume input. An example of grammars that adhere to this condition, are the semi-lexicalized Tree Adjoining Grammars, discussed above. Another example of such a grammar, would be a constraint-based grammar, in which each non-chain-rule is a lexical entry, and each chain-rule branches.

A parser is called *minimal* iff it returns one solution for each possible derivation. As it stands the parser is not minimal in this sense. In fact, if the same word occurs more than once in the input string then we may find a solution several times. The problem comes about because a list is not an appropriate data structure for bags. For example, removing an element from a bag should not be non deterministic, but it is if we use lists. It is straightforward to encode bags with a more appropriate data structure such as the one found in some Prolog libraries, which we will adopt, with slight modifications, in the next subsection. This subsection can be skipped by readers, who indeed believe that such an encoding is possible.

The other subsections discuss ways in which to improve upon the efficiency of the head corner parser. These sections are somewhat technical, and may also be skipped, if the reader is willing to accept that several modifications are possible that improve upon the efficiency of the parser, defined so far.

4.6.1 Representation of bags.

This subsection provides an alternative encoding of 'bags', [3] in order for the 'consume_guide' predicate to be deterministic. The revised parser can then be shown to be minimal.

An empty bag is represented by the constant 'bag'. A nonempty bag consists of three parts: an element, a (representation of a) number indicating how often this element occurs in the bag, and the rest of the bag. Furthermore, the elements in the bag are ordered in some standard order. For example, in Prolog the bag $\{a, b, a, a, b, c\}$ is represented as:

```
bag(a,3,bag(b,2,bag(c,1,bag)))
```

This technique is easily inherited in $\mathcal{R}(\mathcal{L})$ where I encode such an example as:

$$(48)\quad \begin{bmatrix} \mathit{el} : \mathrm{a} \\ \mathit{number} : \begin{bmatrix} s : \begin{bmatrix} s : \begin{bmatrix} s : 0 \end{bmatrix} \end{bmatrix} \end{bmatrix} \\ \mathit{rest} : \begin{bmatrix} \mathit{el} : \mathrm{b} \\ \mathit{number} : \begin{bmatrix} s : \begin{bmatrix} s : 0 \end{bmatrix} \end{bmatrix} \\ \mathit{rest} : \begin{bmatrix} \mathit{el} : \mathrm{c} \\ \mathit{number} : \begin{bmatrix} s : 0 \end{bmatrix} \\ \mathit{rest} : \mathrm{bag} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

For clarity I write such a bag as:

$\langle a/3, b/2, c/1 \rangle$

and use the usual list notation.

The parser is modified as follows. The predicate which calls the parser will contain an extra predicate, *list_to_bag*/2, which encodes a list as a bag. Furthermore this bag is given as the input argument to *parse*/3; the output argument is the empty bag, the constant 'bag'.

(49) $\mathit{guide}(\mathrm{String}, \mathrm{Guide}, \mathrm{bag})$:-
$\quad\quad \mathit{list\_to\_bag}(\mathrm{String}, \mathrm{Guide})$.

The 'consume_guide' predicate is now defined for such bags, as follows:

[3] Based on the file 'bags.pl' of the Quintus library, by Richard O'Keefe.

(50) $\%\mathit{consume\_guide}(+\mathrm{SubBag}, +\mathrm{TotalBag}, ?\mathrm{ComplementBag}).$
$\mathit{consume\_guide}(\langle\rangle, \mathrm{Bag}, \mathrm{Bag}).$
$\mathit{consume\_guide}(\langle \mathrm{El}/0|\mathrm{R}\rangle, \mathrm{Bag}, \mathrm{Rest})$:-
$\quad \mathit{consume\_guide}(\mathrm{R}, \mathrm{Bag}, \mathrm{Rest}).$
$\mathit{consume\_guide}(\langle \mathrm{El}/s(\mathrm{X})|\mathrm{R}\rangle, \langle \mathrm{El}/s(\mathrm{Y})|\mathrm{R}_2\rangle, \mathrm{Rest})$:-
$\quad \mathit{consume\_guide}(\langle \mathrm{El}/\mathrm{X}|\mathrm{R}\rangle, \langle \mathrm{El}/\mathrm{Y}|\mathrm{R}_2\rangle, \mathrm{Rest}).$
$\mathit{consume\_guide}(\langle \mathrm{El}/s(\mathrm{I})|\mathrm{R}\rangle, \langle \mathrm{X}/\mathrm{Y}|\mathrm{Bag}\rangle, \langle \mathrm{X}/\mathrm{Y}|\mathrm{Rest}\rangle)$:-
$\quad \mathit{consume\_guide}(\langle \mathrm{El}/s(\mathrm{I})|\mathrm{R}\rangle, \mathrm{Bag}, \mathrm{Rest}).$

This definition of the predicate 'consume_guide' is deterministic given the first two arguments, and given the fact that the elements of the bag are always constants. The modified version of the parser is minimal.

4.6.2 Indexing of rules

It is possible to use a more clever indexing of rules. Firstly, it is possible to extract the rules from the grammar that can be used to parse some given sentence, before the parser starts properly. That is, for each sentence the parser first selects those rules that possibly could be used in the analysis of that sentence. The parsing algorithm proper then only takes these rules into account when it searches for an applicable rule.

Furthermore, we can get rid of the *consume_guide*/3 predicate. Given the pre-compilation step mentioned above, we can use a slightly different representation of bags where the elements of the bag are not explicitly mentioned, but are implicitly represented by the position in the bag. To make this work we also allow bags where elements occur zero times. For example, given the sentence 'a b b c a a', the corresponding bag will simply be:

$$\langle 3, 2, 1\rangle$$

Furthermore, the bag that consists of two *a*'s is given the representation

$$\langle 2, 0, 0\rangle$$

with respect to that sentence. The idea is that each rule which has been found to be a possible candidate for a given sentence is asserted either as a clause

$$\mathit{predict\_cr}(\mathrm{Head}, \mathrm{M}, \mathrm{Ds}, \mathrm{InBag}, \mathrm{OutBag})$$

or

$$\mathit{predict\_ncr}(\mathrm{M}, \mathrm{Ds}, \mathrm{InBag}, \mathrm{OutBag})$$

where the predicates which made up the body of that predicate are already partially evaluated. For example, given the sentence 'a b b c a a' the indexing step of the parser may find that the (non-chain-rules) 'a', 'b' and 'c' are applicable. These entries are then asserted as:

(51) $predict\_ncr(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \langle \mathrm{s(A)}, \mathrm{B}, \mathrm{C} \rangle, \langle \mathrm{A}, \mathrm{B}, \mathrm{C} \rangle) \text{:-} \phi.$
$predict\_ncr(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \langle \mathrm{A}, \mathrm{s(B)}, \mathrm{C} \rangle, \langle \mathrm{A}, \mathrm{B}, \mathrm{C} \rangle) \text{:-} \phi.$
$predict\_ncr(\mathrm{Goal}, \mathrm{Head}, \mathrm{Ds}, \langle \mathrm{A}, \mathrm{B}, \mathrm{s(C)} \rangle, \langle \mathrm{A}, \mathrm{B}, \mathrm{C} \rangle) \text{:-} \phi.$

The guide is instantiated as follows. The in-part will simply be the bag representation shown above. More precisely for this example:

$\langle s(s(s(0))), s(s(0)), s(0) \rangle$

and the out-part now simply is:

$\langle 0, 0, 0 \rangle$

Note that the 'predict_ncr' and 'head_corner' clauses remain as before.

Schabes (1990) discusses two-step parsing algorithms for lexicalized grammars. In the first step the parser selects the rules which might be applicable in order to parse the given sentence. In the second step the sentence is actually parsed using these rules. Thus, for a LTAG we first select all the initial and auxiliary trees of which the anchors occur in the sentence. This technique is thus quite easily incorporated, using the technique discussed above.

4.6.3 'Order-monotonic' grammars

In some grammars, the string operations that are defined, are not only monotonic with respect to the words they dominate, but also with respect to the order constraints, that are defined between these words ('order-monotonic'). For example, Reape's sequence union operation preserves the linear precedence constraints, that are defined. TAGs do not allow to change the order of two elements once this order has been established. Thus, TAGs are examples of 'order-monotonic' grammars. For that reason, a subsequence check can be used to good effect for the head-driven parser for TAGs. The analysis of verb second in the foregoing section, on the other hand, uses a string operation that does not satisfy this restriction.

For grammars that do satisfy the 'order-monotony' restriction it is possible to extend the top-down prediction possibilities by the incorporation of extra clauses in the 'head_corner' and 'parse' predicates, which check that the phrase that has been analyzed up to that point, can become a sub-string of the input string. To this purpose, the input string is percolated through the parser as an extra argument. Each time a rule has applied the parser checks whether the string derived up to that point can be a subsequence of the input string. The head-corner parser for TAGs defined in the previous section constitutes an example of this technique.

4.6.4 Delaying the extra constraint

In some cases it is very useful to delay the constraint which defines the operations on strings until the parser is finished. For example, if the constraints are disjunctive, then it may be useful to wait as long as possible, before a choice in a certain direction is made. The important information for the parser is percolated anyway through the bag of words; the actual order of the words has (usually) not much influence on other choices of the parser. Hence a lot of uninteresting non determinism can thus be delayed.

As an example, consider a verb which selects a number of arguments, and where furthermore the order of the arguments is free. In effect, the predicate 'cp' non-deterministically produces all possible orders. Suppose that for a given string such a verb is to be parsed, together with two arguments. Then there are six ways to parse this verb-phrase. Only one of these ways corresponds to the order of the input string. The other five possible ways to parse the verb phrase gives rise to re-computation of the other parts of the sentence. If the constraint, which generates the six possible orders, is delayed, then the parser computes with one abstract, unordered, variant, for which at the end of the parse, the proper order can be selected.

In practice this technique increased the efficiency of the parser for some grammars by a factor 3. Clearly this technique is incompatible with the improvement suggested above for order-monotonic grammars.

4.6.5 Memo relations

A possible criticism for the head-driven parser presented here can be formulated as follows. The backtracking search procedure enforces the parser to re-compute possible sub-parses. One of the fundamental principles of parsing, on the other hand, is that you should not do things twice. Hence the head-corner parser does not even adhere to this fundamental principle of parsing.

We should make a methodological distinction between techniques to make the search space as small as possible on the one hand, and techniques to search through the search space as efficiently as possible. The proposed algorithm focuses on ways to make the search space as small as possible. Using tabular methods to assert previous results, in order to prevent double work, constitutes a different dimension along which we should judge parsing algorithms. Techniques to search a given search space can then be 'applied' to a given proof procedure. In the current setting is possible by the introduction of memo-relations. A similar point of view is defended by Leermakers (1991), in the context of LR-parsers.

In the section on memo-relations in chapter 3 I presented a Prolog implementation of so-called 'memo-relations' (this name is inspired by the concept of 'memo-functions' in functional programming languages). The proposed technique to re-use previously established computation results can be applied for the head-driven parser for TAGs as well. In the (small) example grammars I have tested the parser with, this technique did not seem to be practically useful.

For the head-driven parser this technique may not be as useful for the following reason. We might expect that some given noun phrase occurring in the input string will be parsed only once for a given NP goal. In fact however, this noun phrase will be parsed several times for that goal because the input bag of words may be different each time.

For example, if the sentence to be parsed is "the man left yesterday", the memo'ed relation does make a distinction between the goal *np,[s(0),s(0),s(0),0]* (for "the man left") and the goal *np, [s(0),s(0),s(0),s(0)]* (for "the man left yesterday"). This clearly is correct, but less efficient as we might have expected at first.

A possible way to improve upon this, is the following. Each time a result has been found we may generalize this result before the result is asserted in the data-base. For example, if we have shown an np with *[s(0),s(0),s(0),s(0)] - [0,0,s(0),s(0)]* we may generalize this result as the result with *[s(W),s(X),Y,Z]-[W,X,Y,Z]*. The problem with this is that our assumption, that no left-recursion arises, is problematic. That is, we cannot be sure anymore that no duplicate solutions are asserted in the data-base. Of course, it is possible to check before each assertion whether such a result already exists. However, in that case we need to do subsumption checking on the results of the parser, rather than on the goals. This implies much more overhead than before.

4.7 Conclusion

In order for grammars to be reversible, I argued in the beginning of this chapter, that operations on strings, which go beyond concatenation, may be helpful for the following two reasons.

Firstly, such non-concatenative grammars may allow analyses, which reflect more directly the way in which the semantic structures are built up. As a consequence, such grammars may be easier to handle for generation algorithms.

Secondly, I argued that the addition of expressive power, as compared with concatenative grammars, may be useful, in order to obtain grammars, which can effectively be parsed. In the case of concatenative grammars, one is often forced to 'implement' certain discontinuous constituency constructions using the types of rule (empty rules, and non-branching rules) for which there is a danger that the resulting grammar is not effectively parsable anymore.

After a short excursion through a number of proposed extensions to concatenative grammars, I discussed a generalization of these extensions, and provided an example grammar, for a tiny fragment of Dutch.

I then showed that such non-concatenative grammars can be parsed, by a generalization of the left-corner parser: the head-corner parser. This parsing regime was motivated by the desire to use both bottom-up information (the information available in lexical entries), and top-down information (the constraints heads impose upon their arguments — using head-feature percolations.)

A proper superset of Constraint-based Lexicalized Tree Adjoining Grammars can be parsed effectively, using the head corner parser.

Chapter 5

Reversible Constraint-based Machine Translation

This chapter presents the architecture of a *reversible* Machine Translation (MT) system. This architecture is at the core of the MiMo2 prototype which was developed by the author and colleagues at the University of Utrecht (van Noord *et al.*, 1990; van Noord, 1990b; van Noord *et al.*, 1991). The chapter is organized as follows.

In the first section I discuss the notion 'linguistically possible translation', and ways in which we could go about defining linguistically possible translations. A problem, called the subset problem, will be encountered which motivates us to use a specific architecture for MT. In this architecture, which will be presented in some detail in section 5.3, translation is defined by a series of three reversible constraint-based grammars.

In section 5.4 I present some exemplification of the use of constraint-based grammars to define transfer relations between pairs of languages.

In section 5.5 I discuss how to ensure that such transfer components are reversible. It turns out that by requiring that translation is compositional in a sense to be made precise, transfer grammars can be shown to be reversible.

Section 5.6 goes to show that using reversible constraint-based grammars for transfer is expressive enough to define some so-called non-compositional translations, which were problematic for less powerful formalisms. Our conclusion then will be that reversible transfer grammars may be a good compromise between expressive power and computational feasibility.

5.1 Linguistically possible translation

Landsbergen (1984) introduces the notion *linguistically possible translation* which provides a useful methodological concept for work in Machine Translation (see also Landsbergen (1989)). A system which implements the notion 'linguistically possible translation' employs linguistic knowledge only. Clearly, in order to produce the correct or best translation such a system has to be augmented with other (artificial intelligence) components. However, at the current state of technology it seems unrealistic to expect

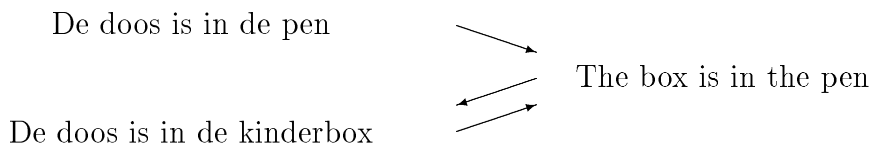


Figure 5.1: Best translation is not symmetric. In the English to Dutch direction, extra-linguistic knowledge dictates the second sentence.

that these other components can be constructed in the near future. Therefore, a more realistic goal for MT consists of the construction of systems implementing the notion 'linguistically possible translation'. Such systems may also be of practical interest, because it may be possible to augment such restricted systems with a component that interacts with the user — for example in the case of difficult disambiguation problems. Thus the user can provide help in order to extract the best translation from the computed set of linguistically possible translations. It is assumed that the overall best translation is in fact to be found (in a significant number of cases) among this set. This chapter therefore will be focused on the notion 'linguistically possible translation', rather than on translation in general.

Consider a class of MT systems which employ linguistic knowledge only. In such a system, a source text is assigned a set of meaning representations according to the rules of the grammar of the source language. On the basis of such a meaning representation the target grammar then produces a set of target sentences for this grammar. Such a system thus produces a set of linguistically *possible* translations. Landsbergen (1984) assumes that the relation 'linguistically possible translation' (lpt) is a symmetric relation. Thus, t_{target} *lpt* t_{source} iff t_{source} *lpt* t_{target}. On the other hand, the relation 'best of linguistically possible translations' is not. For example assume that disambiguation based on knowledge of the world is outside the linguistic components. The asymmetry occurs if the translation of some (unambiguous) source sentence is ambiguous, and where furthermore the 'added' reading of the target sentence is to be preferred for extra-linguistic reasons. An example can be constructed using the famous Bar-Hillel sentence, cf. figure 5.1.

This view of translation is extremely poor. For example, it does not take world knowledge into account as we saw above. Moreover, there are many other factors that could be taken into account in defining linguistically possible translations, e.g. preservation of style, (indirect) speech act, honorifics, etc. It is hoped (and expected) that an approach based on the poor view described can be useful as a basis for future richer views.

An important question of translation is whether there always is a meaning-preserving translation. It may be the case that there are meanings in one language that are not expressible at all in some other language (for some discussion cf. (Katz, 1978; Keenan, 1978)). It may even be the case that one cannot know whether the meaning expressed in two languages is the same (cf. (Quine, 1960)). These are important questions, but the approach outlined here does not depend on how they are answered. Our approach is concerned only with the case where the same meaning can be expressed in both languages. Our question is 'how to describe possible translations', not 'is translation possible'.

The symmetry of the linguistically possible translation relation provides the motivation for *reversible* MT systems (Landsbergen, 1984; Kay, 1984). If the lpt relation from language l_1 to l_2 is in fact the same relation as going from l_2 to l_1, then it seems very natural to try to characterize this relation only once — and to construct a program which is able to compute this relation, given this single characterization, in both directions.

Such an approach has a number of advantages, most of which coincide with the advantages already mentioned in chapter 1. Thus, a reversible architecture has

- theoretical advantages. Given that there *is* only one linguistically possible translation relation between two given languages, then we should describe this relation only once.

- practical advantages. If we are interested in building systems that translate between some given languages then it may be expected that writing a single component for a language pair may in fact be 'cheaper' than writing two separate components.

- methodological advantages. It can be argued that the goal of writing a reversible translation system improves the quality, even in one direction. This point is worked out by (Isabelle, 1989), for the translation between English and French.

5.2 The subset problem

A possible MT system implementing the notion linguistically possible translation may be constructed as the series of two monolingual grammars. It is assumed that each of the monolingual grammars defines a reversible relation between phonological representations and semantic representations. The translation relation simply is the composition of these two monolingual relations. The logical form language is thus used as some sort of interlingua.

In general such an approach faces a problem which has been called the *subset* problem in Landsbergen (1987). Each of the monolingual grammars defines (implicitly) a set of semantic representations. However, in general it need not be the case that for a given semantic representation defined in language a there exists a semantic representation in language b. In general we have the situation as in figure 5.2. In this figure, a grammar for language a relates a set of phonological representations $phon_a$ with a set of semantic representations sem_a; the same holds for a grammar for language b. The set of semantic representations for each language is a subset of the possible sem; however in general sem_a and sem_b are different.

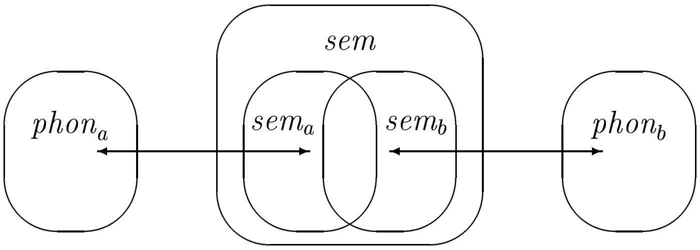


Figure 5.2: The subset problem

The problem can be seen as consisting of two parts. This first part of the subset problem can be characterized as a *difference of coverage* of the source– and target language grammar. The second part constitutes the *logical equivalence* problem.

Difference in coverage. Firstly it may be the case that a given semantic representation simply has no equivalent in the target grammar. Thus a certain meaning cannot be expressed in the target language. In such cases, there is no (linguistically possible) translation available. Thus, in these examples translation does not seem to preserve the meaning (completely). I assume that in general completely different (probably non-monotonic or heuristic) mechanisms are needed for such cases. These examples fall outside the scope of this work.

Logical equivalence. The second reason for the subset problem may be that for a given semantic representation the target language does define a logically equivalent, but syntactically different, semantic representation. This part of the subset problem is an instantiation of the *logical equivalence* problem as discussed in section 1.

There are several ways in which we could go about trying to tackle the subset problem. Solutions to the problem seem to have in common that in some way or other the different grammars of the languages between which translation is defined are put in correspondence, i.e. are *tuned* to each other.

For example, if we are to build a translation system between German and Russian we could construct the monolingual grammars of German and Russian very carefully in such a way that we know that the subset problem does not surface. This approach is worked out in the Rosetta system (Landsbergen, 1987). If we encounter an example in which the German grammar produces a semantic representation for which the Russian grammar does not provide a sentence, we simply change the grammar of German or Russian in such a way that there is a possible translation.

The important counter argument to this approach is, that this leads to a situation in which monolingual grammars are 'impure', whereas from a methodological and practical point of view, we may require that each monolingual grammar should be 'pure', i.e. not influenced by the design of all other monolingual grammars for reasons of modularity. Especially in multi-lingual systems this argument is an important one. Of course, it remains to be seen how important this modularity is in the construction of a practical system.

Instead I propose to tune semantic representations derived by monolingual grammars explicitly. The tuning is defined in an extra component: the *transfer* component. In this case, a translation system between German and Russian is constructed as follows. The monolingual grammars are constructed in a modular way, as desired. For each of the semantic representations (implicitly) defined by the German grammar, we define how these relate to the semantic representations (implicitly) defined by the Russian grammar.

This approach to the subset problem is taken in the MiMo2 system. The semantic representations derived by the monolingual grammars are explicitly tuned to each other by a transfer component. Moreover, this transfer component is defined by a constraint-based grammar (just like the monolingual grammars). For this reason it is very easy to guarantee that the translation relation defined by this system is reversible. Furthermore, if each of the monolingual and transfer grammars is reversible, then so is the translation relation.

The architecture of MiMo2 is defined in more detail in the following section.

5.3 The architecture of MiMo2

In the architecture of MiMo2 to be proposed here, (monolingual) relations between phonological representations and semantic representations are defined by constraint-based grammars of the type introduced in chapter 2. However, constraint-based grammars can also be used to define other relations between (parts of) linguistic signs. In particular it is possible, as discussed by Kay (1984) for FUG, to use constraint-based grammars to define transfer rules. In the model I propose a translation relation between two languages is defined as the composition of three reversible relations. Each of these relations is defined by a constraint-based grammar. The first grammar defines the relation between source language utterances and source language dependent semantic representations. The second grammar defines the relation between source language dependent semantic representations and target language dependent semantic representations, the third grammar defines the relation between target language dependent semantic representations and target language utterances. The resulting MT system is reversible iff each of the grammars is reversible, as I showed in section 1.3).

For example, to compute the relation between Dutch and Spanish phonological representations construct the series of the programs for the Dutch grammar, the Dutch-Spanish transfer grammar and the Spanish grammar. Each translation relation that can be defined is necessarily reversible if each of the grammars that are used defines an reversible relation. See figure 5.3 for an illustration.

The reasons for a constraint-based formalism for transfer rules are the following.

- A constraint-based implementation provides a *declarative* characterization of the transfer relation. This characterization is independent of the actual way in which the relation is computed.

- A single characterization can then be used to compute transfer relations in both

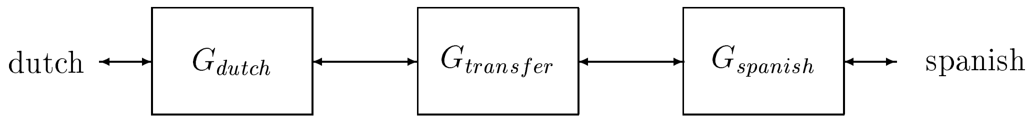


Figure 5.3: Reversible translation system

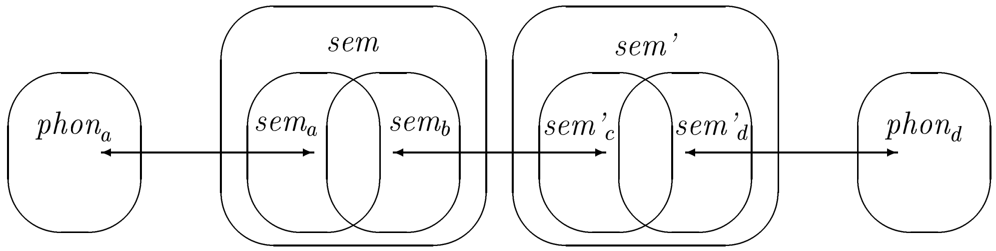


Figure 5.4: The subset problem in a transfer architecture

directions. The constraint-based formalism thus provides for a *reversible* transfer component.

- A constraint-based formalism constitutes a simple, yet very powerful language for the statement of such transfer relations. In section 5.6 I show that transfer relations can be defined to analyze certain non-compositional translations which are problematic for other transfer systems.

In section 5.5 I discuss how to ensure that a transfer grammar is reversible. We show that, as long as translation is compositional in a sense to be made precise, it is possible to guarantee that transfer grammars are reversible. On the other hand such grammars are still powerful enough to handle certain non-compositional translations. For this reason we argue that reversible constraint-based grammars provide for an interesting compromise between expressive power and computability.

As far as the system is concerned, there may be a different logic for natural language semantics for each language. A transfer component for two languages thus functions as an interface to relate the two logics used to define semantic representations with. This makes it possible that grammars are developed quite independently of each other. On the other hand, if languages define similar semantic representations the transfer grammars will generally be simpler and easier to write.

In a transfer model the subset problem, as discussed in the previous section, is in principle present in a slightly different format, because the transfer component need not be 'complete' (cf. figure 5.4). The point at which grammars are connected gives in principle rise to an instantiation of the subset problem. However, in practice it turns out that in the proposed architecture the problem hardly surfaces at all. This is so, because the transfer grammars are explicitly tuned to each of the monolingual grammars. Clearly, that was the reason to have transfer grammars in the first place. Therefore, it seems warranted to neglect this problem.

5.3.1 Other constraint-based approaches to MT

The objective to build a reversible MT system using a series of reversible unification grammars is reminiscent of the CRITTER system (Isabelle *et al.*, 1988), the TFS (Zajac, 1989), the CLE (Alshawi *et al.*, 1991) and ELU (Russell *et al.*, 1991). In CRITTER logic grammars are being used; Emele and Zajac use a type system including an inheritance mechanism to define transfer-like rules. In CLE and ELU the semantic representations defined by monolingual grammars are put in correspondence by special transfer rules, rather than using one formalism for both monolingual and bilingual grammars (as we propose). In section 5.6 I describe an approach to certain non-compositional translation cases which seems not available to the two latter formalisms, but which is available in the MiMo2 architecture.

A somewhat different approach is advocated in Kaplan *et al.* (1989). In that approach a system is described where an LFG grammar for some source language is augmented with equations that define (part of) the target level representations. A generator derives from this partial feature structure a phonological representation according to some LFG grammar of the target language. Instead of a series of three grammars this architecture thus assumes two grammars, one of which defines both the source language and the relation with the target language. The translation relation is not only defined between semantic representations but may relate all levels of representation (*c-structure, f-structure, σ-structure*). Although in this approach monolingual grammars may be used in a bidirectional way it is unclear whether the translation equations can be used bi-directionally. Furthermore, the approach faces problems with certain non-compositional translations as already discussed by the authors and in some more detail in Sadler and Thompson (1991).

Whitelock (1991) discusses a constraint-based approach to MT in which translation relations are exclusively defined by relating lexical entries of the source and target language. A translation in this approach proceeds, somewhat simplified, as follows. A source text is built from a 'bag' (i.e. a multi-set) of source language lexical entries. Each of these lexical entries may be related to lexical entries of the target language, giving rise to a bag of target lexical entries. The target text is computed by generating (non-deterministically) a sentence using this bag of lexical entries (and the combination rules of the target grammar). In relating the lexical entries of different languages, several pieces of information may be put in correspondence; most notably semantic information. The target sentences that are generated use this semantic information. This approach works thanks to a crucial property of the grammars, inherited from UCG (Zeevat *et al.*, 1987): all semantic representations are projected from the lexicon. However, the approach faces severe difficulties for grammars in which semantic information is not always projected from lexical entries. Furthermore, if this approach to translation is generalized in order for other types of information (such as 'style') to be put in correspondence as well, then we need to assume that such information is also projected from lexical entries; it seems problematic to assume that this is always possible or linguistically satisfactory.

5.4 Constraint-based transfer

5.4.1 Simple transfer rules

Semantic representations such as the ones introduced in the first chapter will often be related in a straightforward way to a Dutch equivalent, except for the value of the labels representing content words.

As an example consider a transfer grammar to define the transfer relation between Dutch and English. The semantic representations of English and Dutch are labelled by the labels *gb* and *nl* respectively. We simplify this example by not taking into account properties such as tense and aspect. A rule that translates 'open_fire_on' into 'het_vuur_openen_op' is defined as:

$$(1)\quad sign(\left[\begin{array}{ll} gb: & \left[\begin{array}{l} sort: \text{binary} \\ pred: \text{open\_fire\_on} \\ arg1: \mathrm{G}_1 \\ arg2: \mathrm{G}_2 \\ neg: \text{Neg} \end{array}\right] \\ nl: & \left[\begin{array}{l} sort: \text{binary} \\ pred: \text{het\_vuur\_openen\_op} \\ arg1: \mathrm{N}_1 \\ arg2: \mathrm{N}_2 \\ neg: \text{Neg} \end{array}\right] \end{array}\right]) \texttt{:-}$$

$$sign(\left[\begin{array}{l} gb: \mathrm{G}_1 \\ nl: \mathrm{N}_1 \end{array}\right]),$$

$$sign(\left[\begin{array}{l} gb: \mathrm{G}_2 \\ nl: \mathrm{N}_2 \end{array}\right]).$$

This rule simply states that the translation of an argument structure is composed of the translation of its arguments; furthermore the value for the *neg* attribute is simply defined to be the same for the English and Dutch argument structure representations.

Similar rules can be written for other predicates. If the rule applies to the goal

$$
\texttt{?-}\quad \mathit{sign}\left(\left[\begin{array}{ll}
\mathit{gb}: & \left[\begin{array}{ll}
\mathit{sort}: \text{binary} & \\
\mathit{pred}: \text{open\_fire\_on} & \\
\mathit{arg1}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{soldier}\\ \mathit{num}: \text{pl}\end{array}\right]\\
\mathit{arg2}: & \left[\begin{array}{ll}
\mathit{sort}: \text{modifier} & \\
\mathit{mod}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{columbian}\end{array}\right]\\
\mathit{arg1}: & \left[\begin{array}{ll}
\mathit{sort}: \text{nullary} & \\
\mathit{sort}: \text{modifier} & \\
\mathit{mod}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{prime}\end{array}\right]\\
\mathit{arg1}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{minister}\\ \mathit{num}: \text{sg}\end{array}\right]
\end{array}\right]
\end{array}\right]\\
\mathit{neg}: \text{neg} &
\end{array}\right]
\end{array}\right]_{\mathrm{X}}\right).
$$

the two daughters of the rule will be 'instantiated', and the value of *nl* will be bound to the *nl* values of these daughters; i.e. we obtain the goal:

$$
\texttt{?-}\quad \left[\begin{array}{ll}
\mathit{gb}: \ldots & \\
\mathit{nl}: & \left[\begin{array}{l}
\mathit{sort}: \text{binary}\\
\mathit{pred}: \text{het\_vuur\_openen\_op}\\
\mathit{arg1}: \mathrm{N}_1\\
\mathit{arg2}: \mathrm{N}_2\\
\mathit{neg}: \text{neg}
\end{array}\right]
\end{array}\right]_{\mathrm{X}},
$$

$$
\mathit{sign}\left(\left[\begin{array}{ll}
\mathit{gb}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{soldier}\\ \mathit{num}: \text{pl}\end{array}\right]\\
\mathit{nl}: \mathrm{N}_1 &
\end{array}\right]\right),
$$

$$
\mathit{sign}\left(\left[\begin{array}{ll}
\mathit{gb}: & \left[\begin{array}{ll}
\mathit{sort}: \text{modifier} & \\
\mathit{mod}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{columbian}\end{array}\right]\\
\mathit{arg1}: & \left[\begin{array}{ll}
\mathit{sort}: \text{modifier} & \\
\mathit{mod}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{prime}\end{array}\right]\\
\mathit{arg1}: & \left[\begin{array}{l}\mathit{sort}: \text{nullary}\\ \mathit{pred}: \text{minister}\\ \mathit{num}: \text{sg}\end{array}\right]
\end{array}\right]
\end{array}\right]\\
\mathit{nl}: \mathrm{N}_2 &
\end{array}\right]\right).
$$

An example of the rule for the first daughter will be a bilingual lexical entry and looks as:

$$(2)\quad sign(\left[\begin{array}{l} gb: \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{soldier} \\ num: \text{Num} \end{array}\right], \\ nl: \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{militair} \\ num: \text{Num} \end{array}\right] \end{array}\right]).$$

The second daughter will be translated by a general rule translating argument structures of sort 'modifier' (this rule will be modified later in this section):

$$(3)\quad sign(\left[\begin{array}{l} gb: \left[\begin{array}{l} sort: \text{modifier} \\ mod: \text{G}_{mod} \\ arg1: \text{G}_{arg} \end{array}\right] \\ nl: \left[\begin{array}{l} sort: \text{modifier} \\ mod: \text{N}_{mod} \\ arg1: \text{N}_{arg} \end{array}\right] \end{array}\right]) \texttt{:-}$$

$$sign(\left[\begin{array}{l} gb: \text{G}_{mod} \\ nl: \text{N}_{mod} \end{array}\right]),$$

$$sign(\left[\begin{array}{l} gb: \text{G}_{arg} \\ nl: \text{N}_{arg} \end{array}\right]).$$

The complex English expression 'prime minister' has to be translated as a simple expression in Dutch: 'premier'. This rule can be defined as:

$$(4)\quad sign(\left[\begin{array}{l} nl: \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{premier} \\ num: \text{Num} \end{array}\right] \\ gb: \left[\begin{array}{l} sort: \text{modifier} \\ mod: \text{prime} \\ arg1: \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{minister} \\ num: \text{Num} \end{array}\right] \end{array}\right] \end{array}\right]).$$

where it is assumed that the construction is analyzed in English as an ordinary modified construction (rather than as a single idiomatic expression), and where the semantic representation of the modifier ('prime') takes the semantic representation of the noun as its argument. Note that a similar rule could be written to deal with the 'schimmel - white horse' example. As a result of the rule applications we obtain the following feature structure from which the generator generates the sentence

(5) De militairen hebben het vuur niet geopend op de Columbiaanse premier

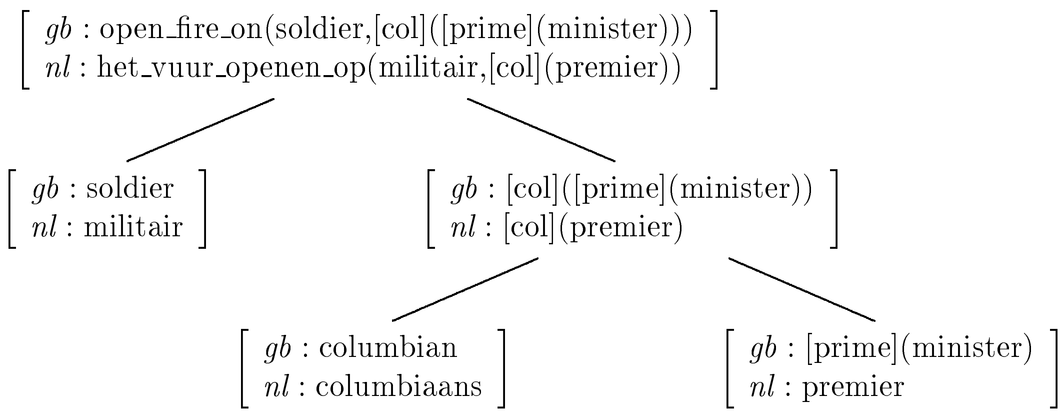


Figure 5.5: Parse tree of transfer example

$$\begin{bmatrix} gb : \ldots \\ nl : \begin{bmatrix} sort : \text{binary} \\ pred : \text{het\_vuur\_openen\_op} \\ arg1 : \begin{bmatrix} sort : \text{nullary} \\ pred : \text{militair} \\ num : \text{pl} \end{bmatrix} \\ arg2 : \begin{bmatrix} sort : \text{modifier} \\ mod : \begin{bmatrix} sort : \text{nullary} \\ pred : \text{columbiaans} \end{bmatrix} \\ arg1 : \begin{bmatrix} sort : \text{nullary} \\ pred : \text{premier} \end{bmatrix} \end{bmatrix} \\ neg : \text{neg} \end{bmatrix} \end{bmatrix}_{\mathrm{X}}$$

The abbreviated 'parse tree' for this example can be shown as in figure 5.5.

In the foregoing examples the relation between semantic representations is rather straightforward. Note however that the full power of a unification grammar can be used to settle more difficult translation cases, because different labels can be used to represent the 'translational syntax'. For instance we can build a tree as value of the label *tree* to represent the derivational history of the translation process. Or we can 'thread' information through different nodes to be able to make translations dependent on each other. Translation parameters such as style and subject field can be percolated as labels of nodes to obtain consistent translations; but these labels themselves need not be translated. Some examples of more interesting translation rules using some of these possibilities are defined in the next sections.

5.4.2 Translating reentrancies

The discussion of reentrancies in relation with the definition of the p-parsing problem in the foregoing chapter (section 2.4) is not without ramifications for the organization of a transfer grammar and the definition of semantic structures. Some constructions such as control verbs and relative clauses may be represented using such reentrancies (essentially as in LFG's f-structure); for example

(6) The soldiers tried to shoot the whisky priest

may be represented by an argument structure where the first argument of 'try' is reentrant with the first argument of 'shoot', cf. :

$$\left[\begin{array}{l} \mathit{sort} : \text{binary} \\ \mathit{pred} : \text{try} \\ \mathit{arg1} : \left[\begin{array}{l} \mathit{sort} : \text{nullary} \\ \mathit{pred} : \text{soldier} \\ \mathit{num} : \text{pl} \end{array}\right]_{\mathrm{X}_1} \\ \mathit{arg2} : \left[\begin{array}{l} \mathit{sort} : \text{binary} \\ \mathit{pred} : \text{shoot} \\ \mathit{arg1} : \mathrm{X}_1 \\ \mathit{arg2} : \left[\begin{array}{l} \mathit{sort} : \text{nullary} \\ \mathit{pred} : \text{whisky\_priest} \\ \mathit{num} : \text{sg} \end{array}\right] \end{array}\right] \end{array}\right]$$

The translation of such argument structures to Dutch equivalents can be defined as in the following rule in matrix notation:

(7) $$\mathit{sign}(\left[\begin{array}{l} \mathit{gb} : \left[\begin{array}{l} \mathit{sort} : \mathit{binary} \\ \mathit{pred} : \text{try} \\ \mathit{arg1} : \mathrm{G}_1 \\ \mathit{arg2} : \left[\ \mathit{arg1} : \mathrm{G}_1\ \right]_{\mathrm{G}_2} \end{array}\right] \\ \mathit{nl} : \left[\begin{array}{l} \mathit{sort} : \mathit{binary} \\ \mathit{pred} : \mathit{proberen} \\ \mathit{arg1} : \mathrm{N}_1 \\ \mathit{arg2} : \left[\ \mathit{arg1} : \mathrm{N}_1\ \right]_{\mathrm{N}_2} \end{array}\right] \end{array}\right]) \text{:-}$$
$$\mathit{sign}(\left[\begin{array}{l} \mathit{gb} : \mathrm{G}_1 \\ \mathit{nl} : \mathrm{N}_1 \end{array}\right]),$$
$$\mathit{sign}(\left[\begin{array}{l} \mathit{gb} : \mathrm{G}_2 \\ \mathit{nl} : \mathrm{N}_2 \end{array}\right]).$$

In this rule the equality representing the control relation is explicitly mentioned for two reasons. The first reason simply is that, given the definition of the p-parsing problem, transfer will not produce anything without explicitly mentioning the reentrancy! The second reason is, that we do not want to translate the two noun phrases in isolation, but rather we want to obtain the same translation for both arguments. These two problems are now explained as follows.

Suppose we did not explicitly mention the reentrancy in the transfer rule. In that case, one of the signs defined by the transfer grammar has the leftmost feature structure in the following figure as the value of its *gb* label (leaving the *sort* attribute out for reasons of space):

$$\left[\begin{array}{ll} \textit{pred}: \text{try} & \\ \textit{arg1}: & \left[\begin{array}{l} \textit{pred}: \text{soldier} \\ \textit{num}: \text{pl} \end{array}\right]_{X_1} \\ \textit{arg2}: & \left[\begin{array}{ll} \textit{pred}: \text{shoot} & \\ \textit{arg1}: & \left[\begin{array}{l} \textit{pred}: \text{soldier} \\ \textit{num}: \text{pl} \end{array}\right]_{X_1} \\ \textit{arg2}: \ldots & \end{array}\right] \end{array}\right] \neq \left[\begin{array}{ll} \textit{pred}: \text{try} & \\ \textit{arg1}: & \left[\begin{array}{l} \textit{pred}: \text{soldier} \\ \textit{num}: \text{pl} \end{array}\right]_{X_1} \\ \textit{arg2}: & \left[\begin{array}{ll} \textit{pred}: \text{shoot} & \\ \textit{arg1}: & \left[\begin{array}{l} \textit{pred}: \text{soldier} \\ \textit{num}: \text{pl} \end{array}\right]_{X_2} \\ \textit{arg2}: \ldots & \end{array}\right] \end{array}\right]$$

On the other hand, the *gb* value of the input for transfer would be the rightmost feature structure. The definition of the *p*-parsing problem (12) requires that the constraints on the *gb* path should be equivalent. However, the second constraint has more solutions than the first constraint (and hence is not equivalent) because the first constraint requires that the paths X_0 *arg1* and X_0 *arg2 arg1* be mapped to the same feature graph, whereas solutions of the second constraint may map these paths to different feature graphs. Hence transfer does not produce a solution if the reentrancy is not 'reproduced' in the transfer grammar.

Even if we were to allow the *p*-parsing problem to produce translations in the case discussed above this would result in some (practical) problems. Suppose indeed that the *p*-parsing problem were relaxed in order to ignore reentrancies. In this case the transfer grammar might be written without mentioning the reentrancy. However, this is also not what we want: the translation of the *arg1* and the embedded *arg1* should clearly be the same. Note that the translation of 'soldier' into Dutch can be both 'soldaat' or 'militair'. If the reentrancy is not mentioned the transfer grammar might also produce the corresponding Dutch semantic representation:

$$\begin{bmatrix} \textit{sort} : \text{binary} \\ \textit{pred} : \text{proberen} \\ \textit{arg1} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{soldaat} \\ \textit{num} : \text{pl} \end{bmatrix} \\ \textit{arg2} : \begin{bmatrix} \textit{sort} : \text{binary} \\ \textit{pred} : \text{neerschieten} \\ \textit{arg1} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{militair} \\ \textit{num} : \text{pl} \end{bmatrix} \\ \textit{arg2} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{whisky\_priester} \\ \textit{num} : \text{sg} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

In most cases the monolingual grammar fails to relate such representations to an utterance, and the system may eventually come up with the appropriate translation as well. However, it is not clear that the monolingual grammar can always be used as a filter for such ill-formed structures. Furthermore such a generate-and-test approach is grossly inefficient.

Note though that unbounded reentrancies can be translated directly in case the reentrancy is between variables. Suppose that the representation for control verbs is not the one shown above, but rather the following (cf. chapter 1):

$$\begin{bmatrix} \textit{sort} : \text{binary} \\ \textit{pred} : \text{try} \\ \textit{arg1} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{soldier} \\ \textit{num} : \text{pl} \\ \textit{index} : \text{I} \end{bmatrix} \\ \textit{arg2} : \begin{bmatrix} \textit{sort} : \text{binary} \\ \textit{pred} : \text{shoot} \\ \textit{arg1} : \begin{bmatrix} \textit{sort} : \text{refer} \\ \textit{index} : \text{I} \end{bmatrix} \\ \textit{arg2} : \begin{bmatrix} \textit{sort} : \text{nullary} \\ \textit{pred} : \text{whisky\_priest} \\ \textit{num} : \text{sg} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

where I introduce the sort 'refer' as a special sort of argument structure. In this case, the reentrancy is only between 'variables' and not between 'structure'. Hence, it suffices to simply state that the value of the attribute 'index' translates as itself, eg.

in bilingual lexical entries which translate nullary argument structures:

$$sign(\left[\begin{array}{ll} gb: & \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{munk} \\ num: \text{Num} \\ index: \text{I} \end{array}\right] \\ nl: & \left[\begin{array}{l} sort: \text{nullary} \\ pred: \text{monnik} \\ num: \text{Num} \\ index: \text{I} \end{array}\right] \end{array}\right]).$$

What remains to be done is to define the following rule for argument structures of type 'refer':

$$sign(\left[\begin{array}{ll} gb: & \left[\begin{array}{l} sort: \text{refer} \\ index: \text{I} \end{array}\right] \\ nl: & \left[\begin{array}{l} sort: \text{refer} \\ index: \text{I} \end{array}\right] \end{array}\right]).$$

Hence if we can limit the need for reentrancies in semantic structures to reentrancies between variables, then the problem disappears.

5.5 Reversible transfer

The formalism also allows transfer grammars that define transfer relations that are not reversible, as is clear from the proof in section 2.5. The objective, though, is to build an *reversible* machine translation system. In this section I will define a simple condition on transfer rules such that a top-down interpreter is guaranteed to terminate for grammars of which the rules satisfy the condition.

The constraint I am about to propose embodies the hypothesis that translation is defined compositionally; i.e. the translation of some structure is defined in terms of the translations of the parts of that structure. In section 5.6 I show that certain types of non-compositional translation can still be handled by a reversible transfer grammar. Therefore I argue that the constraint to be proposed embodies an interesting compromise between expressive power and computability.

Assume the transfer rules define a relation between the paths p and q. I will require that for each rule the value of p of the mother node is strictly larger than the value of p of each of the daughters, and similarly, the value of q of the mother node is strictly larger than the value of q of the mother. I define these sizes in terms of the underlying feature graph models. The size of a feature graph simply is defined as the number of nodes of the graph. For a rule

$$sign(\mathrm{X}_0)\,\text{:-}\,sign(\mathrm{X}_1)\ldots sign(X_n), \phi.$$

I require that for all assignments $\alpha \in \phi^{\mathcal{I}}$, and all j, $1 \leq j \leq n$,

$\alpha(\mathrm{X}_0)^p > \alpha(X_j)^p$
$\alpha(\mathrm{X}_0)^q > \alpha(X_j)^q$

The most straightforward way to satisfy this condition is for a mother node to share proper parts of its p and q values with the p and q values of each of its daughters (making the simplifying assumption that semantic structures are not cyclic). For example, all of the rules given so far satisfy the condition by sharing sub-parts of the value of the *gb* and *nl* attribute. A rule such as the following would fail to meet the condition:

$$(8)\;\; sign(\begin{bmatrix} gb : \begin{bmatrix} pred : \mathrm{kiss} \\ arg1 : \mathrm{G}_1 \\ arg2 : \mathrm{G}_2 \end{bmatrix} \\ nl : \begin{bmatrix} pred : \mathrm{kus} \\ arg1 : \mathrm{N}_1 \\ arg2 : \mathrm{N}_2 \end{bmatrix} \end{bmatrix}_{\mathrm{X}}) \text{:-}$$
$$sign(\begin{bmatrix} gb : \begin{bmatrix} arg1 : \mathrm{G}_1 \\ arg2 : \mathrm{G}_2 \end{bmatrix} \\ nl : \begin{bmatrix} arg1 : \mathrm{N}_1 \\ arg2 : \mathrm{N}_2 \end{bmatrix} \end{bmatrix}_{\mathrm{Y}}).$$

To see that this rule violates the condition, consider the assignment α which assigns both X and Y to a graph whose subgraph at path p is the graph

$$(\mathrm{Z}, \{\mathrm{Z}\ pred\ \mathrm{kus}, \mathrm{Z}\ arg1\ \mathrm{c}_1, \mathrm{Z}\ arg2\ \mathrm{c}_2, \})$$

Another way to understand this, is to see that the feature structure associated with the daughter node, in fact unifies with the feature structure associated with the mother node. Thus, even though the daughter node does not 'mention' the *pred* attribute, this does not mean that there cannot be a value for this attribute. The rules in the previous sections of this chapter all satisfy the condition, as in these rules the daughter nodes are associated with proper sub-parts of the feature structure associated with the mother node.

A top-down interpreter for transfer grammars can be defined as the meta-interpreter defined in chapter 2, repeated here for convenience in figure 5.6. This algorithm thus simply performs a top-down expansion of the input sign. By the size condition we know however that each recursive call constitutes a smaller problem and hence transfer will terminate, given that the ordering on sizes is well-founded. Given the equivalence condition on the p-parsing problem, we know also that there is an upper limit to the possible size of either path p or path q (in principle the interpreter can check at each inference step whether or not it is hypothesizing a further instantiated value of p. Note that the current algorithm does not implement this coherence check; the algorithm can be modified to implement coherence and completeness, as discussed in the preceding chapter.

$$
\begin{array}{l}
\textit{refutation}(\mathrm{Goal})\!:\!- \\
\qquad \textit{rule}(\mathrm{Goal}, \mathrm{Ds}), \\
\qquad \textit{refutations}(\mathrm{Ds}). \\
\\
\textit{refutations}(\langle\rangle). \\
\textit{refutations}(\langle \mathrm{H}|\mathrm{T}\rangle)\!:\!- \\
\qquad \textit{refutation}(\mathrm{H}), \\
\qquad \textit{refutations}(\mathrm{T}).
\end{array}
$$

Figure 5.6: Meta interpreter for $\mathcal{R}(\mathcal{L})$-grammars

5.6 Reversible Transfer of Context-sensitive Translations

The purpose of this section is to show that the formalism proposed for transfer is more powerful than some previously defined transfer formalisms such as the CAT framework (Arnold *et al.*, 1986; van Noord *et al.*, 1989), but also some of the constraint-based transfer formalisms (Alshawi *et al.*, 1991; Russell *et al.*, 1991), even taking into account the constraint on transfer grammars defined in the previous section. Furthermore, this extra power is required to handle some non-compositional translations. As in monolingual uses of constraint-based grammars we may percolate all kinds of information in the feature structures, for example to define context-sensitive translations.

In formalisms such as CAT, a transfer rule essentially translates a tree by translating parts of the tree recursively. The rules thus always operate on parts of the input object; this input object cannot be modified. This leads to complex rules for the treatment of context-sensitive translations. For example, the English adjective 'strong' is normally translated into 'sterk' in Dutch. In the case of 'strong criticism', however, the translation has to be 'scherpe kritiek' (sharp criticism). Assuming that this regularity has to be treated in transfer, the only possible way to obtain this result in CAT is to define a transfer rule that translates the structure in which both 'strong' and 'criticism' occur. This is problematic in cases where this larger structure contains other parts that have to be translated as well; the translation of these other parts can however also be irregular. This leads to special rules to handle the combination of such irregular cases (see also Arnold *et al.* (1988)). It may also be problematic if the representation of the adjective and the noun do not appear as sisters in the representation but may be arbitrarily far away from each other (e.g. because other adjectives intervene). In that case a rule has to be written for each different possibility. In cases where there is no limit to the distance even this escape will not work.

The current framework allows a compositional treatment of such context-sensitive cases because we simply can percolate information using the constraints. The validity of this claim is established with the following an example. First, recall that the argument structures for noun phrases such as 'very strong whisky', look as follows:

$$\begin{bmatrix} \mathit{sort} : \text{modifier} \\ \mathit{mod} : \begin{bmatrix} \mathit{sort} : \text{modifier} \\ \mathit{mod} : \begin{bmatrix} \mathit{sort} : \text{nullary} \\ \mathit{pred} : \text{very} \end{bmatrix} \\ \mathit{arg1} : \begin{bmatrix} \mathit{sort} : \text{nullary} \\ \mathit{pred} : \text{strong} \end{bmatrix} \end{bmatrix} \\ \mathit{arg1} : \begin{bmatrix} \mathit{sort} : \text{nullary} \\ \mathit{pred} : \text{whisky} \\ \mathit{num} : \text{sg} \end{bmatrix} \end{bmatrix}$$

The basic rule to translate argument structures of sort 'modifier' is defined as in the following rule (this rule replaces the preceding rule for modifier structures). In this rule the information of what the 'head' of a modified structure is, is percolated through the features 'gbhead' and 'nlhead'. This value will then be equated with the features 'gbheaded' and 'nlheaded' that are associated with the modifiers.

$$(9)\quad \mathit{sign}(\begin{bmatrix} \mathit{nl} : \begin{bmatrix} \mathit{sort} : \text{modifier} \\ \mathit{mod} : \text{N}_{mod} \\ \mathit{arg1} : \text{N}_{arg} \end{bmatrix} \\ \mathit{gb} : \begin{bmatrix} \mathit{sort} : \text{modifier} \\ \mathit{mod} : \text{G}_{mod} \\ \mathit{arg1} : \text{G}_{arg} \end{bmatrix} \\ \mathit{gbhead} : \text{G}_{head} \\ \mathit{nlhead} : \text{N}_{head} \\ \mathit{gbheaded} : \text{G}_{headed} \\ \mathit{nlheaded} : \text{N}_{headed} \end{bmatrix}) :\!-$$

$$\mathit{sign}(\begin{bmatrix} \mathit{nl} : \text{N}_{mod} \\ \mathit{gb} : \text{G}_{mod} \\ \mathit{gbheaded} : \text{G}_{head} \\ \mathit{nlheaded} : \text{N}_{head} \end{bmatrix}),$$

$$\mathit{sign}(\begin{bmatrix} \mathit{nl} : \text{N}_{arg} \\ \mathit{gb} : \text{G}_{arg} \\ \mathit{gbhead} : \text{G}_{head} \\ \mathit{nlhead} : \text{N}_{head} \\ \mathit{gbheaded} : \text{G}_{headed} \\ \mathit{nlheaded} : \text{N}_{headed} \end{bmatrix}).$$

The rule translating nouns such as 'whisky' is defined as:

(10) $$sign(\left[\begin{array}{l} nl : \left[\begin{array}{l} sort : \text{nullary} \\ pred : \text{whisky} \\ agr : \text{Agr} \end{array}\right] \\ gb : \left[\begin{array}{l} sort : \text{nullary} \\ pred : \text{whisky} \\ agr : \text{Agr} \end{array}\right] \\ nlhead : \text{whisky} \\ gbhead : \text{whisky} \end{array}\right]).$$

Now we are ready to define a special rule for the translation of 'strong' into 'scherp' if the adjective is headed by the noun 'criticism'/'kritiek'.

(11) $$sign(\left[\begin{array}{l} nl : \left[\begin{array}{l} sort : \text{nullary} \\ pred : \text{scherp} \end{array}\right] \\ gb : \left[\begin{array}{l} sort : \text{nullary} \\ pred : \text{strong} \end{array}\right] \\ nlheaded : \text{kritiek} \\ gbheaded : \text{criticism} \end{array}\right]).$$

Hence, the parse tree of the translation of 'very strong unmotivated criticism' can be given as in figure 5.7. We thus showed how the use of 'extra' constraints allows for a compositional (in fact reversible) treatment of context-sensitive translations.

5.7 Conclusion

This chapter presented the reversible architecture of the MiMo2 prototype. The architecture is motivated by considering the 'subset problem', as discussed by Landsbergen (1987). The linguistically possible translation relation between two languages is defined by a series of reversible constraint-based grammars. I showed how constraint-based grammars can be used to define transfer rules. By requiring that transfer rules are 'compositional' in a certain sense made precise, we showed that grammars can be guaranteed to be reversible. I also showed that certain non-compositional translations can still be handled in the framework I proposed. This motivated the claim that reversible transfer grammars provide an interesting compromise between expressive power and computational feasibility.

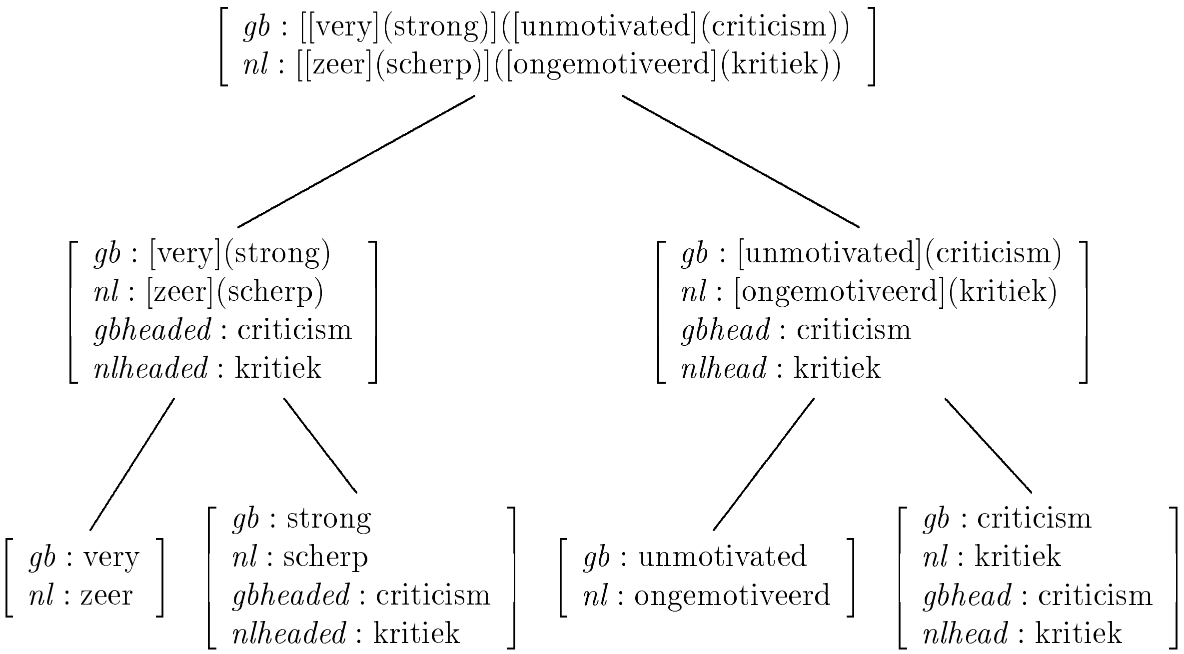


Figure 5.7: Translating ‘very strong unmotivated criticism’

Summary

Introduction

Constraint-based grammars are often used for natural language processing. One of the interesting properties of a constraint-based grammar is, that such a grammar is completely *declarative*: it does not enforce a specific processing regime, but allows various parsing and generation algorithms. The order of processing is independent on the result of the computation.

For that reason, such declarative grammars can, at least in principle, be used both for parsing and generation. However, from a somewhat more practical point of view, several problems must be solved once the same grammar is used for both parsing and generation. Some of these problems are the subject of this thesis. For example, it turns out that applying a 'naive' processing strategy for the purpose of generation, gives rise to severe problems for grammars which are not written with the purpose of generation in mind.

The first chapter of the thesis provides the motivation for reversible grammars, and clarifies some of the fundamental issues involved. A grammar is seen as a device which defines a relation between form and meaning. Form and meaning are represented by phonological structures and semantic structures. A parsing algorithm computes for a given phonological structure its corresponding semantic structure. A generation algorithm computes the relation between form and meaning in the opposite direction.

An *effectively reversible* (or *reversible* for short) grammar is defined as a grammar which defines an effectively reversible relation (between form and meaning). A binary relation is effectively reversible if and only if it can be computed in both directions by a program which terminates for all inputs. Hence, a grammar is reversible if both parsing and generation is guaranteed to terminate (for all inputs).

Reversible grammars are interesting for linguistic, technological and psychological reasons. Linguistically it can be argued that a single language should be described by a single grammar. Furthermore, if such a single grammar is to be used as a 'theory' about the language, it can be argued that it should be possible to check the predictions the theory makes about the language, and hence the grammar should be reversible.

From a language technological point of view, I argue in section 1.1 that reversible grammars make it easier to build good natural language processing systems. It is sometimes thought that grammars which are used for parsing only, can be very liberal in that such grammars may allow ungrammatical sentences (as these are not expected to occur in the input). I argue, on the contrary, that this type of overgeneration is bad, because it generally leads to false ambiguities. Hence, even if one is only interested in building a parsing system, it may be the case that a reversible grammar is a good method to obtain such a system. Moreover, it may be easier to build a single, reversible grammar, than two separate grammars.

In section 1.1 it is discussed whether humans base their language production and language understanding on a single body of grammatical knowledge. This claim would explain why humans speak the same language they understand and vice versa. It is argued that observed differences in language understanding and production may be due to differences at another level of cognitive behavior, rather than to differences in the grammatical component.

An important goal of the thesis is to improve upon existing parsing and generation techniques along the following two dimensions. Firstly, an important motivation is to extend the types of grammar for which the proposed parsing and generation techniques are applicable. Secondly, the parsing and generation techniques are motivated from a linguistic perspective. It is hoped that such a 'linguistic' motivation for deduction improves the efficiency of parsing and generation as compared with non-linguistic deduction techniques.

The formalism: $\mathcal{R}(\mathcal{L})$

Chapter 2 defines a formalism, called $\mathcal{R}(\mathcal{L})$, which is in several respects typical for formalisms in use in computational linguistics. It can be characterized as a constraint-based formalism, where the constraints are restricted to the path-equations known from PATR II. Unless PATR II, the formalism does not prescribe that phrases are built by concatenation. Hence the formalism is comparable to pure Prolog, but instead of first-order terms, the data-structures of the formalisms are feature structures (defined by path equations). Furthermore, the formalism is defined within a very general setting, provided by the work of Höhfeld and Smolka (1988). This provides for more powerful constraints to be added to the formalism, without affecting some of the properties of the formalism, if the necessary constraint-solving techniques are available for these more powerful constraints.

The resulting formalism, $\mathcal{R}(\mathcal{L})$, is used in the thesis as a language to define grammars with, but moreover as a language to define meta-interpreters in. A grammar of $\mathcal{R}(\mathcal{L})$ is a definite clause specification of the relation *sign*/1. A simplified clause of a grammar might read as follows:

(1) $\textit{sign}(\mathrm{M})$:-
$\quad\textit{sign}(\mathrm{D}_1)$,
$\quad\textit{sign}(\mathrm{D}_2)$,
$\quad\mathrm{M}\ \textit{cat} \doteq \mathrm{s}$,
$\quad\mathrm{D}_1\ \textit{cat} \doteq \mathrm{np}$,
$\quad\mathrm{D}_2\ \textit{cat} \doteq \mathrm{vp}$,
$\quad\mathrm{D}_1\ \textit{agr} \doteq \mathrm{D}_2\ \textit{agr}$,
$\quad\mathrm{M}\ \textit{phon in} \doteq \mathrm{D}_1\ \textit{phon in}$,

$$\mathrm{D}_1\ \mathit{phon}\ \mathit{out} \doteq \mathrm{D}_2\ \mathit{phon}\ \mathit{in},$$
$$\mathrm{D}_2\ \mathit{phon}\ \mathit{out} \doteq \mathrm{M}\ \mathit{phon}\ \mathit{out},$$
$$\mathrm{M}\ \mathit{sem} \doteq \mathrm{D}_1\ \mathit{sem}.$$

Such a clause will generally be written in matrix notation as follows:

$$(2)\quad sign(\begin{bmatrix} syn : \mathrm{s} \\ phon : \mathrm{P}_0 - \mathrm{P} \\ sem : \mathrm{S} \end{bmatrix}) \texttt{:-}$$
$$sign(\begin{bmatrix} syn : \mathrm{np} \\ agr : \mathrm{A} \\ phon : \mathrm{P}_0 - \mathrm{P}_1 \end{bmatrix}),$$
$$sign(\begin{bmatrix} syn : \mathrm{np} \\ agr : \mathrm{A} \\ sem : \mathrm{S} \\ phon : \mathrm{P}_1 - \mathrm{P} \end{bmatrix}).$$

The procedural semantics of the formalism is comparable to Prolog's proof procedure. Thus, refutation of a goal proceeds in a top-down manner. A left-most computation rule is assumed, such that the leftmost atom of a goal is reduced first. Furthermore, the search tree is traversed in a depth-first backtrack manner. Alternative proof strategies for $\mathcal{R}(\mathcal{L})$ grammars are defined later as meta-interpreters in $\mathcal{R}(\mathcal{L})$.

In the case we use $\mathcal{R}(\mathcal{L})$ for grammars, the p-parsing problem for a path p is defined as follows. Assume the input for parsing is a feature structure with phonological representation ϕ as the value of the attribute *phon*. The answers to the *phon*-parsing problem will be those compatible signs in the grammar which have ϕ as the value of their *phon* attribute. Similarly, assume the input for generation is some feature structure with semantic representation σ as the value of the attribute *sem*. The answers to the *sem*-parsing problem are those compatible signs in the grammar which have *sem* as their semantic structure.

It is shown that in general the p-parsing problem for $\mathcal{R}(\mathcal{L})$ grammars is not solvable, by showing that an undecidable problem, Post's Correspondence Problem, can be defined in a $\mathcal{R}(\mathcal{L})$ grammar. It can thus be concluded that in the general case, grammars of $\mathcal{R}(\mathcal{L})$ are not reversible.

In practice however, the grammars computational linguists tend to write *are* reversible. For this reason an important task consists in the construction of proof procedures which solve the p-parsing problem for the grammars typically encountered. This is the goal of the third and fourth chapter of the thesis. In these chapters generation- and parsing techniques are presented, which are more generally applicable than some competing techniques. Moreover these techniques are motivated linguistically in that certain *head-driven* and *lexicalistic* aspects of most modern grammars are exploited in those techniques. It is hoped that the linguistic foundation of these processing techniques leads to an increased efficiency.

Semantic-head-driven Generation

In the third chapter, I discuss the generation problem for $\mathcal{R}(\mathcal{L})$ grammars, and propose a semantic-head-driven generation technique which I claim is superior to some other approaches to generation that have been proposed previously. In order for this technique to be useful, the semantic structures should be defined in a lexical and semantic-head-driven fashion. Typically, the semantics of a phrase is a further instantiation of the semantics of the semantic-head of the phrase. And furthermore, the semantics of the other parts of the phrase, is determined by the semantic-head of the phrase. Semantic-head-driven generation proceeds by *predicting* the lexical head of a given phrase. This lexical head is then *connected* upward to the goal by successive rule applications.

Clauses which have a daughter with identical semantics as the mother node, are called *chain-rules*, and are, for the purpose of the generation meta-interpreter, represented as:

$$cr(\mathrm{Head}, \mathrm{Mother}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle) \,\text{:-}\, \phi.$$

On the other hand, rules without a head (such as lexical entries) are called *non-chain-rules* and are represented as

$$ncr(\mathrm{Mother}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle) \,\text{:-}\, \phi.$$

The following meta-interpreter in figure 1 defines a simple instantiation of the semantic-head-driven processing strategy. In chapter 3 several improvements and variations of this strategy are discussed.

The semantic-head-driven strategy is compared with the top-down oriented approaches of Wedekind (1988) and Dymetman and Isabelle (1988), and the chart-based approach of Shieber (1988). Semantic-head-driven generation is more general than those top-down oriented approaches as it handles certain linguistically motivated left-recursive analyses of subcategorization. On the other hand semantic-head-driven generation must be favored over Shieber's chart-based generator because it allows certain non-compositional analyses, such as those proposed for idiomatic constructions.

An important problem for semantic-head-driven generation are linguistic analyses which are based on a threading implementation of head-movement. I provide an analysis of verb-second in Dutch along those lines, which is problematic for semantic-head-driven generation (because it violates the assumptions about the construction of the semantic structures mentioned above). A further problem for semantic-head-driven generation is exemplified by an analysis of English raising-to-object constructions.

Although I do provide some ad-hoc solutions to these problems, it can be argued that a more general solution is available only, if we allow grammars in which phrases can be built with other operations than concatenation. For example, raising-to-object constructions can be analyzed using Bach's 'wrap' mechanism. The resulting analysis is not problematic for semantic-head-driven generation. On the other hand, such more powerful operations on strings lead to an increased burden on the parser.

$bug(\text{Goal})$:-
    $predict\_head(\text{Goal}, \text{Lex})$,
    $sem\_head(\text{Lex}, \text{Goal})$.

$sem\_head(\text{Goal}, \text{Goal})$.
$sem\_head(\text{Head}, \text{Goal})$:-
    $cr(\text{Head}, \text{Mother}, \text{Others})$,
    $bug\_ds(\text{Others})$,
    $sem\_head(\text{Mother}, \text{Goal})$.

$bug\_ds(\langle\rangle)$.
$bug\_ds(\langle \text{H}|\text{T}\rangle)$:-
    $bug(\text{H})$,
    $bug\_ds(\text{T})$.

$predict\_head(\text{Goal}, \text{Head})$:-
    $\text{Goal}\ sem \doteq \text{Head}\ sem$,
    $ncr(\text{Head}, \text{Ds})$,
    $bug\_ds(\text{Ds})$.

Figure 1: A simple version of the semantic-head-driven meta-interpreter. In chapter 3 some improvements of this algorithm are discussed.

Head-corner parsing

In chapter 4 I discuss some proposals for string operations beyond concatenation, such as Pollard's proposal for an incorporation of several head-wrapping operations in a GPSG; Joshi's Tree Adjoining Grammars; and Reape's proposal for an incorporation of sequence-union constraints in an HPSG. I define a head-driven parsing algorithm, called 'head-corner' parsing, for a class of grammars in which strings are constructed in a *linear* and *non-erasing* fashion. Linearity (or 'non-copying') requires that a given rule only introduces some constant number of terminal symbols. A grammar rule is non-erasing, if the terminal symbols associated with the mother node is a superset of those associated with the daughter nodes. The proposals mentioned above are in this class. The head-corner parser thus is applicable for a superset of concatenative grammars, whereas most other parsers are only applicable for concatenative grammars.

Head-corner parsing is a parsing technique which proceeds *head-driven* and *bidirectionally*; both in the sense that the parser does not proceed from left-to-right, nor does it proceed either bottom-up or top-down. As the parser proceeds head-driven it is possible to exploit the usual percolation of syntactic features between the head-daughter and the mother node, in order to improve upon the goal-directedness of the algorithm. Furthermore, such an order of processing exploits the fact that heads determine what other categories may occur.

In order for the head-corner parser to be generally applicable for linear and non-erasing grammars, the input string is used as a guide during the parsing process. In contrast to parsers for concatenative grammars, the elements from this guide are not necessarily removed from left to right, but can be removed in any order. In other words, the guide functions as a set, rather than a stack. To understand the basics of this algorithm, assume that rules are represented as follows.

For simplicity, assume that all terminal symbols are introduced on rules without daughters (lexical entries), and that all rules with daughters have a head (in chapter 4 no such simplification is assumed). A clause with a head daughter is represented as:

$$cr(\mathrm{Head}, \mathrm{Mother}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle)\,\texttt{:-}\,\phi.$$

Lexical entries dominating the terminal symbol Word are represented as:

$$lex(\mathrm{Word}, \mathrm{Mother})\,\texttt{:-}\,\phi.$$

Finally assume that the predicate *head*/2 defines the information which is shared between a syntactic head and its mother node. This information might for example be defined as HPSG's Head Feature Principle. Given these assumptions, a simple version of the head-corner parser can be defined as in figure 2.

The remainder of chapter 4 discusses extensions and variations of this general scheme. As an example, it is shown how constraint-based versions of Lexicalized Tree Adjoining Grammars can be parsed by a variant of the head-corner parser. An important reduction of the search space for head-corner parsing can be achieved for grammars in which the operations on strings are monotonic with respect to the ordering of the terminal symbols they define. Such a monotonicity property is exhibited by TAGs. It is shown how a simple check improves the efficiency of the parser.

(3) $parse(\mathrm{Goal}, \mathrm{P}_0, \mathrm{P})$:-
$\quad head(\mathrm{Lex}, \mathrm{Goal}),$
$\quad del(\mathrm{Word}, \mathrm{P}_0, \mathrm{P}_1),$
$\quad lex(\mathrm{Word}, \mathrm{Lex}),$
$\quad head\_corner(\mathrm{Head}, \mathrm{Goal}, \mathrm{P}_1, \mathrm{P}).$

$del(\mathrm{El}, \langle \mathrm{El} | \mathrm{T} \rangle, \mathrm{T}).$
$del(\mathrm{El}, \langle \mathrm{H} | \mathrm{T} \rangle, \langle \mathrm{H} | \mathrm{T}_2 \rangle)$:-
$\quad del(\mathrm{El}, \mathrm{T}, \mathrm{T}_2).$

$parse\_ds(\langle\rangle, \mathrm{P}, \mathrm{P}).$
$parse\_ds(\langle \mathrm{H} | \mathrm{T} \rangle, \mathrm{P}_0, \mathrm{P})$:-
$\quad parse(\mathrm{H}, \mathrm{P}_0, \mathrm{P}_1),$
$\quad parse\_ds(\mathrm{T}, \mathrm{P}_1, \mathrm{P}).$

$head\_corner(\mathrm{X}, \mathrm{X}, \mathrm{P}, \mathrm{P}).$
$head\_corner(\mathrm{Small}, \mathrm{Big}, \mathrm{P}_0, \mathrm{P})$:-
$\quad cr(\mathrm{Small}, \mathrm{Mid}, \mathrm{Ds}),$
$\quad parse\_ds(\mathrm{Ds}, \mathrm{P}_0, \mathrm{P}_1),$
$\quad head\_corner(\mathrm{Mid}, \mathrm{Big}, \mathrm{P}_1, \mathrm{P}).$

Figure 2: A simple version of the head-corner parser. In chapter 4 I discuss a more general version, and some improvements.

Reversible MT

The final chapter of the thesis proposes an application of reversible grammars. It is shown how a series of reversible grammars can be used to implement Landsbergen's notion of a linguistically possible translation. In this proposal reversible monolingual grammars of two languages are interfaced using a third, reversible, transfer grammar. A transfer grammar defines a relation between language specific semantic structures. Such a transfer grammar can be defined as a grammar of $\mathcal{R}(\mathcal{L})$ as well.

As an example of a rule of such a transfer grammar, consider the translation of the following pair of sentences of English and Dutch:

(4) John bevalt Mary
John is-liked-by Mary
Mary likes John

It is assumed that the Dutch semantic structure takes John to be the first, and Mary the second argument. In the English semantic structure the situation is reversed. The following transfer rule might be written to translate such structures into each other. The labels *nl* and *gb* refer to the Dutch resp. the English semantic structure.

$$(5)\quad sign(\begin{bmatrix} nl: & \begin{bmatrix} pred: \text{bevalt} \\ arg1: \text{Nl}_1 \\ arg2: \text{Nl}_2 \end{bmatrix} \\ gb: & \begin{bmatrix} pred: \text{likes} \\ arg1: \text{Gb}_1 \\ arg2: \text{Gb}_2 \end{bmatrix} \end{bmatrix}) \text{:-}$$

$$sign(\begin{bmatrix} nl: \text{Nl}_2 \\ gb: \text{Gb}_1 \end{bmatrix}),$$

$$sign(\begin{bmatrix} nl: \text{Nl}_1 \\ gb: \text{Gb}_2 \end{bmatrix}),$$

It is discussed how a simple constraint on such transfer grammars can be defined, to guarantee that transfer is effective. This constraint requires that for a given transfer rule between languages l_1 and l_2, the value of the l_1 attribute of the mother node is strictly larger than each of the values for this attribute of the daughters. The same condition holds for the l_2 attribute. It can easily be shown that for a grammar whose rules adhere to this condition, termination is guaranteed for all inputs.

It might be expected that such a constraint reduces the expressive power of a transfer grammar. It is shown, by means of an example, that powerful feature percolations can be used to implement certain 'non-compositional' translation cases. It can thus be argued that reversible transfer grammars in fact constitute an interesting compromise between expressive power and computability.

Samenvatting in het Nederlands

Omkeerbaarheid in natuurlijke-taalverwerking

Introductie

'Constraint-based' grammatica's worden vaak gebruikt bij het automatisch verwerken van natuurlijke taal. Declarativiteit is een interessante eigenschap van zulke grammatica's. De grammatica schrijft niet voor hoe de verwerking van de grammatica plaats moet vinden. Verschillende parseer- en genereertechnieken kunnen worden gebruikt, omdat de volgorde van de berekeningen onafhankelijk is van het uiteindelijke resultaat van de berekeningen. Om deze redenen kunnen constraint-based grammatica's in principe zowel voor parseren, als voor genereren gebruikt worden. Praktisch gezien dienen echter nog verschillende problemen opgelost te worden voor dit ideaal verwezenlijkt kan worden. Het blijkt bijvoorbeeld dat een naïeve genereerstrategie problemen oplevert voor grammatica's die in eerste instantie niet voor generatie bedoeld waren. Sommige van deze problemen vormen het onderwerp van deze dissertatie.

Het eerste hoofdstuk bespreekt de motieven voor omkeerbare grammatica's. Een grammatica definieert een relatie tussen vorm en betekenis. Vorm en betekenis worden in zo'n grammatica gerepresenteerd met fonologische en semantische strukturen. Een parseeralgoritme berekent voor een gegeven fonologische structuur de bijbehorende semantische strukturen. Een genereeralgoritme berekent voor een gegeven semantische structuur de bijbehorende fonologische strukturen.

Een *effectief omkeerbare* (of kortweg *omkeerbare*) grammatica wordt gedefinieerd als een grammatica die een effectief omkeerbare relatie definieert (tussen fonologische en semantische strukturen). Een relatie is effectief omkeerbaar dan en slechts dan als deze relatie kan worden uitgerekend in beide richtingen door een programma dat gegarandeerd termineert voor iedere mogelijke invoer.

Omkeerbare grammatica's zijn interessant om taalkundige, taaltechnologische en psychologische redenen. De taalkundige relevantie blijkt uit de stelling dat één taal door één grammatica beschreven dient te worden. Bovendien kan worden beargumenteerd dat de voorspellingen die deze grammatica doet over de taal controleerbaar moeten zijn. In een omkeerbare grammatica is het controleerbaar op welke manier semantische en fonologische structuren zijn gerelateerd.

In sectie 1.1. wordt aangetoond dat het gebruik van omkeerbare grammatica's het bouwen van goede taalverwerkende systemen kan vergemakkelijken. Men denkt soms dat grammatica's die slechts voor het ontleden van taal gebruikt worden wat vrijer kunnen zijn en ongrammaticale zinnen kunnen toestaan. In praktische toepassingen komen ongrammaticale zinnen misschien toch niet voor. In de meeste gevallen leidt dit type overgeneratie echter ook tot foute ambiguiteiten: het systeem geeft meerdere analyses voor een zin, terwijl in feite slechts één analyse juist is. Dus ook al is men alleen geïnteresseerd in het ontwerpen van een parseersysteem dan kan het gebruik van omkeerbare grammatica's toch interessant zijn.

In dezelfde sectie wordt de vraag opgeworpen of het taalgebruik van mensen gebaseerd is op een enkele taalkundige component. Dit zou verklaren waarom mensen altijd dezelfde taal spreken als ze verstaan, en omgekeerd. Het kan worden beargumenteerd dat de verschillen tussen receptief en produktief taalgebruik misschien verklaard kunnen worden door verschillen op een ander nivo van cognitief gedrag, en niet door verschillen in de taalkundige component.

Een belangrijk doel van de dissertatie is om bestaande parseer- en genereerprocedures te verbeteren wat betreft hun toepasbaarheid en hun taalkundige relevantie. Dus een belangrijk doel is om parseer- en genereertechnieken te ontwerpen die voor meer soorten grammatica's toepasbaar zijn. Ten tweede dienen zulke technieken taalkundig gemotiveerd te kunnen worden, in de hoop dat zulke taalkundige motivatie uiteindelijk leidt tot een verbeterde efficiëntie.

Het formalisme $\mathcal{R}(\mathcal{L})$

In hoofdstuk 2 wordt een formalisme gedefinieerd dat in verschillende opzichten representatief is voor de formalismen die binnen de computationele taalkunde gebruikt worden. Het formalisme, $\mathcal{R}(\mathcal{L})$, is gebaseerd op 'constraints'. De constraints die gebruikt worden zijn de 'path equations' (pad vergelijkingen) bekend van PATR II. Echter in het formalisme wordt niet voorgeschreven dat zinsdelen worden opgebouwd door middel van concatenatie, maar wordt de mogelijkheid opengehouden dat andere methoden gebruikt worden om zinsdelen samen te voegen. Het formalisme is dus te zien als een variant van puur Prolog, waarbij feature-strukturen de plaats van eerste orde termen overnemen. Bovendien wordt het formalisme gedefinieerd binnen het algemene kader van Höhfeld and Smolka (1988), waardoor het mogelijk is om krachtigere constraints aan het formalisme toe te voegen zonder dat bepaalde eigenschappen van het systeem verloren gaan, zolang de juiste 'constraint-solving' technieken hiervoor beschikbaar zijn.

Het resulterende formalisme wordt in de dissertatie zowel gebruikt om grammatica's in te definiëren, als om 'meta-interpreters' in te definiëren. Een grammatica in $\mathcal{R}(\mathcal{L})$ is een 'definite clause' specificatie van de relatie *sign*/1. Een simpel voorbeeld van een regel uit zo'n grammatica is de volgende clause:

(1) *sign*(M) :-
 sign(D_1),
 sign(D_2),
 M *cat* $\doteq$ s,
 D_1 *cat* $\doteq$ np,
 D_2 *cat* $\doteq$ vp,

$$
\begin{aligned}
&\mathrm{D}_1\ agr \doteq \mathrm{D}_2\ agr,\\
&\mathrm{M}\ phon\ in \doteq \mathrm{D}_1\ phon\ in,\\
&\mathrm{D}_1\ phon\ out \doteq \mathrm{D}_2\ phon\ in,\\
&\mathrm{D}_2\ phon\ out \doteq \mathrm{M}\ phon\ out,\\
&\mathrm{M}\ sem \doteq \mathrm{D}_1\ sem.
\end{aligned}
$$

Zo'n regel wordt in de matrix notatie als volgt weergegeven:

$$
(2)\quad sign(\begin{bmatrix} syn : \mathrm{s} \\ phon : \mathrm{P}_0 - \mathrm{P} \\ sem : \mathrm{S} \end{bmatrix}) \text{:-}
$$

$$
sign(\begin{bmatrix} syn : \mathrm{np} \\ agr : \mathrm{A} \\ phon : \mathrm{P}_0 - \mathrm{P}_1 \end{bmatrix}),
$$

$$
sign(\begin{bmatrix} syn : \mathrm{np} \\ agr : \mathrm{A} \\ sem : \mathrm{S} \\ phon : \mathrm{P}_1 - \mathrm{P} \end{bmatrix}).
$$

De procedurele semantiek van $\mathcal{R}(\mathcal{L})$ is vergelijkbaar met Prologs zoekmethode. Dat wil dus zeggen dat een refutatie van een 'goal' plaats vindt volgens de 'top-down' methode. De 'computation rule' die wordt gebruikt selecteert steeds het meest linkse atoom. Daarnaast wordt de zoekruimte doorzocht met een 'depth-first back-track' strategie. Andere zoekprocedures worden later gedefinieerd in $\mathcal{R}(\mathcal{L})$ als meta-interpretators.

Indien $\mathcal{R}(\mathcal{L})$ gebruikt wordt om grammatica's in te definiëren dan wordt het p-parseer probleem als volgt gedefinieerd. Stel, de invoer voor het parseren is een feature structuur waarvan de fonologische structuur ϕ is als de waarde van het attribuut *phon*. De antwoorden op het *phon*-parseerprobleem zijn dan al die signs uit de grammatica die niet in tegenspraak zijn met de invoer, en bovendien ook ϕ als waarde voor hun *phon* attribuut hebben. Op dezelfde manier worden de antwoorden op het *sem*-parseerprobleem gedefinieerd als die signs uit de grammatica die compatibel zijn met de invoer, en dezelfde waarde hebben voor het *sem* attribuut.

In hoofdstuk 2 wordt aangetoond dat in zijn algemeenheid het p-parseer probleem voor $\mathcal{R}(\mathcal{L})$ grammatica's onoplosbaar is, omdat het mogelijk is een onbeslisbaar probleem (Post's Correspondence Problem) te coderen als een $\mathcal{R}(\mathcal{L})$ grammatica. Hieruit volgt dan ook onmiddellijk dat $\mathcal{R}(\mathcal{L})$ grammatica's in zijn algemeenheid niet omkeerbaar zijn.

In de praktijk is het echter zo dat de grammatica's die door computationeel taalkundigen geschreven worden wel degelijk omkeerbaar zijn. Om deze reden is het dus een belangrijke taak bewijsprocedures te construeren die het p-parseer probleem oplossen voor de grammatica's die in de praktijk gebruikt blijken te worden. Deze taak beslaat het belangrijkste deel van deze dissertatie (de hoofdstukken 3 en 4). In deze hoofdstukken worden genereer- en parseertechnieken ontwikkeld die ruimer toepasbaar zijn dan sommige andere methoden. Bovendien worden deze technieken gemotiveerd vanuit een taalkundig oogpunt omdat bepaalde 'head-driven' en lexicalistische eigenschappen van moderne grammatica's uitgebuit worden, in de hoop dat dit zal leiden tot een grotere efficientie.

'Semantic-head-driven' generatie

In het derde hoofdstuk bediscussieer ik het generatie probleem (i.e. het *sem*-parseerprobleem). Ik stel een generatieprocedure voor die gestuurd wordt door de notie 'semantisch hoofd'. Ik laat zien dat deze methode voordelen biedt boven sommige andere methoden die eerder voorgesteld zijn. De techniek is bruikbaar in het geval semantische strukturen gedefinieerd worden op een lexicale en door het semantisch-hoofd gestuurde manier. De semantische structuur van een zinsdeel is in die methode een verdere instantiatie van de semantische structuur van zijn hoofd. Bovendien bepaalt het hoofd van een zinsdeel de semantische strukturen van de andere onderdelen van dat zinsdeel.

'Semantic-head-driven' generatie gaat in zijn werk door middel van het voorspellen van het semantische hoofd van een zinsdeel, waarna door middel van het toepassen van de regels van de grammatica getracht wordt dit hoofd in verbinding te brengen met het oorspronkelijke doel.

Regels waarbij één der dochters dezelfde semantische structuur als de moeder heeft, worden 'chain-rules' genoemd, en worden ten behoeve van de meta-interpretator gerepresenteerd als:

$$cr(\mathrm{Hoofd}, \mathrm{Moeder}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle)\,\text{:-}\,\phi.$$

Regels zonder hoofd (zoals lexicale elementen) worden 'non-chain-rules' genoemd en worden gerepresenteerd als:

$$ncr(\mathrm{Moeder}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle)\,\text{:-}\,\phi.$$

De meta-interpretator in figuur 1 definieert een eenvoudige instantiatie van de generatiemethode die gestuurd wordt door semantische hoofden.

De 'semantic-head-driven' generatiemethode wordt vergeleken met de methoden van Wedekind (1988) en Dymetman and Isabelle (1988) die top-down georienteerd zijn, en de op Earley gebaseerde methode van Shieber (1988). 'Semantic-head-driven' generatie is algemener dan top-down generatie omdat het bepaalde taalkundig gemotiveerde links-recursieve analyses aankan die problematisch zijn voor top-down generatie. Daarnaast staat 'semantic-head-driven' generatie bepaalde analyses toe van idiomatische constructies die problematisch zijn voor de Earley-methode van Shieber.

Een belangrijk probleem voor 'semantic-head-driven' generatie zijn bepaalde taalkundige analyses die gebruik maken van een 'threading' implementatie van 'head-movement'. Ik geef zo'n analyse voor 'verb-second' in het Nederlands, die problematisch is voor dit type generatie (omdat de constructie van de semantische structuur niet plaatsvindt zoals hierboven werd aangegeven). Nog een ander probleem wordt besproken aan de hand van een analyse van Engelse 'raising-to-object' constructies.

Hoewel het mogelijk is enige 'ad-hoc' oplossingen te ontwerpen, zal ik beargumenteren dat een algemenere oplossing voor deze problemen slechts mogelijk is wanneer

$$
\begin{array}{l}
\mathit{bug}(\mathrm{Doel})\,\texttt{:-} \\
\quad \mathit{voorspel\_hoofd}(\mathrm{Doel}, \mathrm{Lex}), \\
\quad \mathit{sem\_hoofd}(\mathrm{Lex}, \mathrm{Doel}). \\[1em]
\mathit{sem\_hoofd}(\mathrm{Doel}, \mathrm{Doel}). \\
\mathit{sem\_hoofd}(\mathrm{Hoofd}, \mathrm{Doel})\,\texttt{:-} \\
\quad \mathit{cr}(\mathrm{Hoofd}, \mathrm{Moeder}, \mathrm{Andere}), \\
\quad \mathit{bug\_ds}(\mathrm{Andere}), \\
\quad \mathit{sem\_hoofd}(\mathrm{Moeder}, \mathrm{Doel}). \\[1em]
\mathit{bug\_ds}(\langle\rangle). \\
\mathit{bug\_ds}(\langle \mathrm{H}|\mathrm{S}\rangle)\,\texttt{:-} \\
\quad \mathit{bug}(\mathrm{H}), \\
\quad \mathit{bug\_ds}(\mathrm{S}). \\[1em]
\mathit{voorspel\_hoofd}(\mathrm{Doel}, \mathrm{Hoofd})\,\texttt{:-} \\
\quad \mathrm{Doel}\ \mathit{sem} \doteq \mathrm{Hoofd}\ \mathit{sem}, \\
\quad \mathit{ncr}(\mathrm{Hoofd}, \mathrm{Ds}), \\
\quad \mathit{bug\_ds}(\mathrm{Ds}).
\end{array}
$$

Figuur 1: Een eenvoudige variant van ‘semantic-head-driven’ generatie. In hoofdstuk 3 worden verschillende verbeteringen van deze eenvoudige variant besproken.

aangenomen kan worden dat zinsdelen geconstrueerd kunnen worden door middel van andere operaties dan concatenatie. Bijvoorbeeld, de Engelse raising-to-object constructies kunnen geanalyseerd worden met Bach's 'wrapping' mechanisme. De resulterende analyse is niet problematisch voor 'semantic-head-driven' generatie. De mogelijkheid van krachtigere operaties op strings vergroot de belasting voor parseertechnieken.

'Head-corner' parseren

In hoofdstuk 4 bespreek ik een aantal voorstellen voor het combineren van zinsdelen die uitgaan boven concatenatie, zoals bijvoorbeeld Pollard's 'head-wrapping' operaties binnen een GPSG; Joshi's Tree Adjoining Grammars, Reape's incorporatie van 'sequence-union constraints' binnen een HPSG, of 'liberation' binnen een categoriale grammatica.

Ik definieer een parseerprocedure, 'head-corner parsing' genaamd, voor een klasse van grammatica's waarbij zinsdelen geconstrueerd worden op een 'non-erasing' en lineaire manier. Een grammaticaregel is lineair wanneer de regel slechts een constant aantal terminale symbolen introduceert. Een regel is 'non-erasing' wanneer de terminale symbolen geassocieerd met de dochterknopen een deelverzameling is van de terminale symbolen geassocieerd met de moederknoop. De genoemde voorstellen behoren tot deze klasse van grammatica's. De head-corner parser is dus bruikbaar voor een verzameling grammatica's die de concatenatieve grammatica's omvat.

Head-corner parsing is een parseertechniek die zowel hoofdgestuurd als bidirectioneel te werk gaat (niet van links naar rechts, noch top-down of bottom-up). Omdat de parser hoofdgestuurd te werk gaat is het mogelijk de gebruikelijke percolatie van eigenschappen tussen de moederknoop en het hoofd uit te buiten, om zodoende de doelgerichtheid van de parser te vergroten. Daarnaast gebruikt deze parseervolgorde het feit dat het hoofd normaliter bepaalt uit welke andere onderdelen een zinsdeel kan bestaan. Als je weet wat het hoofd van een zinsdeel is, weet je dus welke andere onderdelen je nog moet tegenkomen.

Om de head-corner parser bruikbaar te laten zijn voor lineaire en 'non-erasing' grammatica's wordt de invoerzin als een 'gids' gebruikt tijdens het parseren. Echter, de manier waarop elementen uit de gids geconsumeerd worden is niet van links naar rechts, maar onbepaald. De gids is dus geen stapel maar een verzameling.

Om te begrijpen hoe de parser werkt nemen we aan dat regels als volgt gedefinieerd zijn. Alle terminale symbolen worden geintroduceerd op een regel zonder dochter ('lexical entry'), en elke regel met dochters heeft ook een hoofd (in hoofdstuk 4 worden deze aannames niet gemaakt). Een regel met een hoofd wordt gerepresenteerd als:

$$cr(\mathrm{Hoofd}, \mathrm{Moeder}, \langle \mathrm{D}_1 \ldots \mathrm{D}_n \rangle) :\!- \phi.$$

Lexicale elementen worden gerepresenteerd als:

$$lex(\mathrm{Woord}, \mathrm{Moeder}) :\!- \phi.$$

(3) $\mathit{parse}(\mathrm{Doel}, \mathrm{P}_0, \mathrm{P})$:-
    $\mathit{hoofd}(\mathrm{Lex}, \mathrm{Doel})$,
    $\mathit{del}(\mathrm{Woord}, \mathrm{P}_0, \mathrm{P}_1)$,
    $\mathit{lex}(\mathrm{Woord}, \mathrm{Lex})$,
    $\mathit{head\_corner}(\mathrm{Hoofd}, \mathrm{Doel}, \mathrm{P}_1, \mathrm{P})$.

$\mathit{del}(\mathrm{El}, \langle \mathrm{El}|\mathrm{T}\rangle, \mathrm{T})$.
$\mathit{del}(\mathrm{El}, \langle \mathrm{H}|\mathrm{T}\rangle, \langle \mathrm{H}|\mathrm{T}_2\rangle)$:-
    $\mathit{del}(\mathrm{El}, \mathrm{T}, \mathrm{T}_2)$.

$\mathit{parse\_ds}(\langle\rangle, \mathrm{P}, \mathrm{P})$.
$\mathit{parse\_ds}(\langle \mathrm{H}|\mathrm{T}\rangle, \mathrm{P}_0, \mathrm{P})$:-
    $\mathit{parse}(\mathrm{H}, \mathrm{P}_0, \mathrm{P}_1)$,
    $\mathit{parse\_ds}(\mathrm{T}, \mathrm{P}_1, \mathrm{P})$.

$\mathit{head\_corner}(\mathrm{X}, \mathrm{X}, \mathrm{P}, \mathrm{P})$.
$\mathit{head\_corner}(\mathrm{Klein}, \mathrm{Groot}, \mathrm{P}_0, \mathrm{P})$:-
    $\mathit{cr}(\mathrm{Klein}, \mathrm{Mid}, \mathrm{Ds})$,
    $\mathit{parse\_ds}(\mathrm{Ds}, \mathrm{P}_0, \mathrm{P}_1)$,
    $\mathit{head\_corner}(\mathrm{Mid}, \mathrm{Groot}, \mathrm{P}_1, \mathrm{P})$.

Figuur 2: Een eenvoudige variant van de head-corner parser. In hoofdstuk 4 wordt een algemenere variant gedefinieerd, en worden enkele verbeteringen besproken.

Daarnaast definieert het predikaat *hoofd*/2 de informatie die hoofden gemeen hebben met hun moederknoop. Een simpele versie van de head-corner parser kan nu gedefinieerd worden zoals in figuur 2.

Het vervolg van hoofdstuk 4 bespreekt enkele uitbreidingen en verbeteringen van de head-corner parser. Bij wijze van uitgewerkt voorbeeld laat ik zien hoe Lexicalized Tree Adjoining Grammars geparseerd kunnen worden met behulp van een variant van de head-corner parser. Een belangrijke reductie van de zoekruimte kan worden bereikt voor grammatica's waarbij de operaties op strings monotoon zijn met betrekking tot de volgorde van de terminale symbolen. TAG's bezitten deze monotonieeigenschap bijvoorbeeld. Ik laat zien hoe door middel van een simpele uitbreiding de head-corner parser deze eigenschap kan benutten, om zo de efficiëntie van de parser te vergroten.

Omkeerbaar automatisch vertalen

Het laatste hoofdstuk van de dissertatie stelt een toepassing voor van omkeerbare grammatica's. Ik laat zien hoe een serie omkeerbare grammatica's gebruikt kunnen worden om het door Landsbergen geintroduceerde begrip 'taalkundig mogelijke vertaling' te implementeren. In dit voorstel wordt een omkeerbare grammatica gebruikt om omkeerbare monolinguale grammatica's aan elkaar te koppelen. Zo'n 'transfer'-

grammatica definieert een relatie tussen taalspecifieke semantische structuren, en kan ook als een $\mathcal{R}(\mathcal{L})$ grammatica geformuleerd worden.

Als een voorbeeld van een regel uit zo'n transfer grammatica, beschouw het volgende vertaalvoorbeeldje:

(4) John bevalt Mary
John is-liked-by Mary
Mary likes John

Ik neem aan dat in de Nederlandse semantische structuur 'John' het eerste argument, en 'Mary' het tweede argument is. In het Engels is het precies omgekeerd. Om zulke semantische structuren aan elkaar te koppelen is de volgende regel mogelijk, waarbij de attributen *nl* en *gb* respectievelijk de Nederlandse en Engelse semantische structuur representeren.

$$(5)\quad sign(\begin{bmatrix} nl: \begin{bmatrix} pred: \text{bevalt} \\ arg1: \text{Nl}_1 \\ arg2: \text{Nl}_2 \end{bmatrix} \\ gb: \begin{bmatrix} pred: \text{likes} \\ arg1: \text{Gb}_1 \\ arg2: \text{Gb}_2 \end{bmatrix} \end{bmatrix}) \texttt{:-}$$

$$sign(\begin{bmatrix} nl: \text{Nl}_2 \\ gb: \text{Gb}_1 \end{bmatrix}),$$

$$sign(\begin{bmatrix} nl: \text{Nl}_1 \\ gb: \text{Gb}_2 \end{bmatrix}),$$

In hoofdstuk 5 wordt een eenvoudige conditie op regels uit zo'n transfergrammatica opgesteld, die garandeert dat de transfergrammatica omkeerbaar is. Kort gezegd komt deze conditie er op neer, dat in een transferregel tussen de talen l_1 en l_2, de waarde van het l_1-attribuut 'groter' is dan elk van de waardes voor dit attribuut van de dochters. Voor l_2 geldt dezelfde conditie. Het kan gemakkelijk aangetoond worden dat voor grammatica's waarbij elke regel aan de conditie voldoet een eenvoudige top-down procedure gegarandeerd termineert.

Zo'n conditie op mogelijke transferregels verkleint natuurlijk de kracht van het formalisme. Ik laat echter zien dat het nog steeds mogelijk is krachtige featurepercolaties te gebruiken om bijvoorbeeld context-gevoelige vertaalvoorbeelden te kunnen analyseren. Ik beargumenteer daarom dat omkeerbare transfergrammatica's een interessant compromis vormen tussen expressieve kracht, en berekenbaarheid.

Bibliography

Anne Abeille. Parsing french with tree adjoining grammar: some linguistic accounts. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

Hiyan Alshawi and Stephen G. Pulman. Ellipsis, comparatives, and generation. In Hiyan Alshawi, editor, *The Core Language Engine*, pages 251–275. ACL-MIT press, 1992. chapter 13.

Hiyan Alshawi, David Carter, and Manny Rayner. Translation by quasi logical form transfer. In *29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, 1991.

Douglas E. Appelt. Bidirectional grammars and the design of natural language generation systems. In *Theoretical Issues in Natural Language Processing 3*, pages 206–212, New Mexico State University, 1987.

K.R. Apt. Introduction to logic programming. Technical Report CS-R8741, Centrum voor Wiskunde en Informatica (Centre for Mathematics and Computer Science), Amsterdam, 1987. also appears in: Handbook of Theoretical Computer Science (J. van Leeuwen, managing editor), North Holland.

Doug Arnold, Steven Krauwer, Mike Rosner, Louis des Tombe, and Nino Varile. The CAT framework in eurotra: A theoretically committed notation for MT. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING)*, Bonn, 1986.

Doug Arnold, Steven Krauwer, Louisa Sadler, and Louis des Tombe. Relaxed compositionality in machine translation. In *Proceedings of the Second International Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages*, Pittsburgh, 1988. Carnegie Mellon University.

Emmon Bach. Control in Montague grammar. *Linguistic Inquiry*, 10:515–553, 1979.

Hans Ulrich Block. Compiling trace & unification grammar for parsing and generation. In *Proceedings of the ACL workshop Reversible Grammar in Natural Language Processing*, Berkeley, 1991.

Joan Bresnan, editor. *The Mental Representation of Grammatical Relations.* MIT Press, 1982.

Stephan Busemann. *Generierung natuerlicher Sprache mit Generalisierten Phrasenstruktur-Grammatiken.* PhD thesis, University of the Saar, 1990. Also available as TU Berlin, Dept. of Computer Science, KIT report 87.

Jonathan Calder, Mike Reape, and Henk Zeevat. An algorithm for generation in unification categorial grammar. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, pages 233–240, Manchester, 1989.

Alain Colmerauer. *PROLOG II: Manuel de réference et modèle théorique.* Groupe d'Intelligence Artificielle, Faculté, des Sciences de Luminy, Marseille, France, 1982.

Luis Damas and Giovanni B. Varile. Constraint logic grammars. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

David Dowty. Towards a minimalist theory of syntactic structure. In *Proceedings of the Symposium on Discontinuous Constituency*, ITK Tilburg, 1990.

Marc Dymetman and Pierre Isabelle. Reversible logic grammars for machine translation. In *Proceedings of the Second International Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages*, Pittsburgh, 1988.

Marc Dymetman, Pierre Isabelle, and Francois Perrault. A symmetrical approach to parsing and generation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

Marc Dymetman. Inherently reversible grammars, logic programming and computability. In *Proceedings of the ACL workshop Reversible Grammar in Natural Language Processing*, Berkeley, 1991.

Jay Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 14, 1970. Also reprinted in Grosz *et al.* (1986).

Arnold Evers. *The Transformational Cycle in Dutch and German.* PhD thesis, Rijksuniversiteit Utrecht, 1975.

J.E. Fenstad, P-K Halvorsen, T. Langholm, and J. van Benthem. *Situations, Language and Logic.* Reidel, Dordrecht, 1987.

Clare Gardent and Agnes Plainfossé. Generating from a deep structure. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

Gerald Gazdar, Ewan Klein, Geoffrey Pullum, and Ivan Sag. *Generalized Phrase Structure Grammar.* Blackwell, 1985.

Dale Douglas Gerdemann. *Parsing and Generation of Unification Grammars.* PhD thesis, University of Illinois at Urbana-Champaign, 1991. Cognitive Science technical report CS-91-06 (Language Series).

Barbara Grosz, Karen Sparck Jones, and Bonny Lynn Webber, editors. *Readings in Natural Language Processing.* Morgan Kaufmann, 1986.

Andrew Haas. A parsing algorithm for unification grammar. *Computational Linguistics*, 15(4), 1989.

Per-Kristian Halvorsen and Ronald Kaplan. Projections and semantic description in lexical-functional grammar. In *Proceedings of the International Conference on Fifth Generation Computer Systems*, Tokyo, 1988. Institute for New Generation Computer Technology.

J. Haviland. Guugu yimidhirr. In R. Dixon and B.Blake, editors, *Handbook of Australian Languages.* Benjamins Amsterdam, 1979.

Mark Hepple. *The Grammar and Processing of Order and Dependency: a Categorial Approach.* PhD thesis, University of Edinburgh, 1990.

Susan Hirsch. P-PATR: A compiler for unification-based grammars. In Veronique Dahl and Patrick Saint-Dizier, editors, *Natural Language Understanding and Logic Programming II.* North Holland, 1988.

Jack Hoeksema. Complex predicates and liberation in dutch and english. *Linguistics and Philosophy*, 14:661–710, 1991.

Markus Höhfeld and Gert Smolka. Definite relations over constraint languages. Technical Report 53, LILOG IBM, Stuttgart, 1988. to appear in Journal of Logic Programming.

John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages and Computation.* Addison Wesley, 1979.

Riny Huybrechts. The weak inadequacy of context-free phrase structure grammars. In Ger de Haan, Mieke Trommelen, and Wim Zonneveld, editors, *Van Periferie naar Kern.* Foris, 1984.

Pierre Isabelle, Marc Dymetman, and Elliott Macklovitch. CRITTER: a translation system for agricultural market reports. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

Pierre Isabelle. Towards reversible MT systems. In *MT Summit II*, Munich, 1989.

R.S. Jackendoff. *X' Syntax: A Study of Phrase Structure.* MIT press, Cambridge Mass., 1977.

Mark Johnson. Parsing with discontinuous constituents. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, 1985.

Mark Johnson. *Attribute Value Logic and the Theory of Grammar*. Center for the Study of Language and Information Stanford, 1988.

A.K. Joshi, L.S. Levy, and M. Takahashi. Tree adjunct grammars. *Journal Computer Systems Science*, 10(1), 1975.

Ronald Kaplan, Klaus Netter, Jürgen Wedekind, and Annie Zaenen. Translation by structural correspondences. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, Manchester, 1989.

J. Katz. Effability and translation. In Guenthner and Guenthner-Reutter, editors, *Meaning and Translation*. Duckworth, 1978.

Martin Kay. Syntactic processing and functional sentence perspective. In *Theoretical Issues in Natural Language Processing — supplement to the Proceedings*, pages 12–15, Cambridge Massachusetts, 1975.

Martin Kay. Functional unification grammar: A formalism for machine translation. In *Proceedings of the 10th International Conference on Computational Linguistics and the 22nd Annual Meeting of the Association for Computational Linguistics (COLING)*, Stanford, 1984.

Martin Kay. Parsing in functional unification grammar. In David R. Dowty, Lauri Karttunen, and Arnold M. Zwicky, editors, *Natural Language Parsing*. Ellis Horwood/Wiley, 1985. reprinted in Grosz *et al.* (1986).

Martin Kay. Head driven parsing. In *Proceedings of Workshop on Parsing Technologies*, Pittsburgh, 1989.

E. Keenan. Some logical problems in translation. In Guenthner and Guenthner-Reutter, editors, *Meaning and Translation*. Duckworth, 1978.

Margaret King, editor. *Machine Translation, the State of the Art*. Edinburgh University Press, 1987.

Jan Koster. Dutch as an SOV language. *Linguistic Analysis*, 1, 1975.

Robert Kowalski. *Logic for problem solving*. Elsevier Science Publishing, 1979.

Jan Landsbergen. Isomorphic grammars and their use in the Rosetta translation system, 1984. Paper presented at the tutorial on Machine Translation, Lugano; also appears in King (1987).

Jan Landsbergen. Montague grammar and machine translation. In Pete Whitelock, Mary McGee Wood, Harold Somers, Rod Johnson, and Paul Bennett, editors, *Linguistic Theory & Computer Applications*. Academic Press, London, 1987.

Jan Landsbergen. Kunnen machines vertalen?, 1989. Oratie Rijksuniversiteit Utrecht, ISBN 90-9003208-8.

Rene Leermakers. Non-deterministic recursive ascent parsing. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin, 1991.

Y. Matsumoto, H. Tanaka, H. Hirakawa, H. Miyoshi, and H. Yasukawa. BUP: a bottom up parser embedded in Prolog. *New Generation Computing*, 1(2), 1983.

J. McCloskey. A VP in a VSO language? In Gerald Gazdar, Ewan Klein, and Geoffrey K. Pullum, editors, *Order, Concord and Constituency*. Foris, 1983.

Kathleen McKeown. *Text Generation*. Cambridge University Press, Cambridge, England, 1985.

Robert C. Moore. Unification-based semantic interpretation. In *27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

Michael Moortgat. A fregean restriction on meta-rules. In *Proceedings of NELS 14*, University of Massachusetts, Amherst, 1984.

John Nerbonne. Feature-based disambiguation. In M. Rossner, C.J. Rupp, and R. Johnson, editors, *Constraint Propagation, Linguistic Description, and Computation*, Lugano, 1991. Istituto Dalle Molle IDSIA, Working Paper No. 5.

John Nerbonne. Constraint-based semantics. Technical Report RR-92-18, DFKI Saarbrücken, 1992.

Günter Neumann and Gertjan van Noord. Self–monitoring with reversible grammars. In *Proceedings of the 14th International Conference on Computational Linguistics (COLING)*, Nantes, 1992.

Fernando C.N. Pereira and Stuart M. Shieber. *Prolog and Natural Language Analysis*. Center for the Study of Language and Information Stanford, 1987.

Fernando C.N. Pereira and David Warren. Definite clause grammars for language analysis - a survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13, 1980. reprinted in Grosz *et al.* (1986).

Fernando C.N. Pereira and David Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, Cambridge Massachusetts, 1983.

Fernando C.N. Pereira. Extraposition grammars. *Computational Linguistics*, 7(4), 1981.

Carl Pollard and Ivan Sag. *Information Based Syntax and Semantics, Volume 1*. Center for the Study of Language and Information Stanford, 1987.

Carl Pollard. *Generalized Context-Free Grammars, Head Grammars, and Natural Language*. PhD thesis, Stanford, 1984.

Carl Pollard. Categorial grammar and phrase structure grammar: An excursion on the syntax-semantics frontier. In R. T. Oehrle, Emmon Bach, and D. Wheeler, editors, *Categorial Grammars and Natural Language Structures*. Reidel, 1988.

W. V. Quine. *Word and Object*. MIT Press, 1960.

Mike Reape. A logical treatment of semi-free word order and bounded discontinuous constituency. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, UMIST Manchester, 1989.

Mike Reape. Getting things in order. In *Proceedings of the Symposium on Discontinuous Constituency*, ITK Tilburg, 1990.

Mike Reape. Parsing bounded discontinuous constituents: Generalisations of some common algorithms. In *Proceedings of the first CLIN dag*. OTS RUU Utrecht, 1991.

Herbert Ruessink and Gertjan van Noord. Remarks on the bottom-up generation algorithm, 1989. unpublished paper.

C.J. Rupp. Constraint propagation and semantic representation. In M. Rossner, C.J. Rupp, and R. Johnson, editors, *Constraint Propagation, Linguistic Description, and Computation*, Lugano, 1991. Istituto Dalle Molle IDSIA, Working Paper No. 5.

Graham Russell, Susan Warwick, and John Carroll. Asymmetry in parsing and generating with unification grammars: Case studies from ELU. In *28th Annual Meeting of the Association for Computational Linguistics*, University of Pittsburgh, 1990.

Graham Russell, Afzal Ballim, Dominique Estival, and Susan Warwick. A language for the statement of binary relations over feature structures. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin, 1991.

Louisa Sadler and Henry S. Thompson. Structural non-correspondence in translation. In *Fifth Conference of the European Chapter of the Association for Computational Linguistics*, Berlin, 1991.

Giorgio Satta and Oliviero Stock. Head-driven bidirectional parsing. a tabular method. In *Proceedings of the Workshop on Parsing Technologies*, pages 43–51, Pittsburgh, 1989.

Giorgio Satta and Oliviero Stock. Bidirectional context-free grammar parsing for natural language processing, 1991. Ms. IRST Trento.

Yves Schabes. *Mathematical and Computational Aspects of Lexicalized Grammars.* PhD thesis, University of Pennsylvania, 1990.

Stuart M. Shieber, Hans Uszkoreit, Fernando C.N. Pereira, J. Robinson, and M. Tyson. The formalism and implementation of PATR-II. In B. J. Grosz and M. E. Stickel, editors, *Research on Interactive Acquisition and Use of Knowledge.* SRI report, 1983.

Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. A semantic-head-driven generation algorithm for unification based formalisms. In *27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

Stuart M. Shieber, Gertjan van Noord, Robert C. Moore, and Fernando C.N. Pereira. Semantic-head-driven generation. *Computational Linguistics*, 16(1), 1990.

Stuart M. Shieber. Using restriction to extend parsing algorithms for complex-feature-based formalisms. In *23th Annual Meeting of the Association for Computational Linguistics*, Chicago, 1985.

Stuart M. Shieber. A uniform architecture for parsing and generation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

Stuart M. Shieber. *Parsing and Type Inference for Natural and Computer Languages.* PhD thesis, Stanford University, 1989. SRI International Technical note 460.

Klaas Sikkel and Rieks op den Akker. Head-corner chart parsing. In *Proceedings Computing Science in the Netherlands (CSN '92)*, Utrecht, 1992.

Gert Smolka. Feature constraint logics for unification grammars. Technical report, IBM Wissenschaftliches Zentrum, Institut für Wissensbasierte Systeme, 1989. IWBS Report 93.

Marc Steedman. Dependency and coordination in the grammar of dutch and english. *Language*, 61, 1985.

Henry Thompson. Strategy and tactics: a model for language production. In *Papers from the Thirteenth Regional Meeting*, Chicago Linguistic Society, 1977.

Henry S. Thompson. Generation and translation - towards a formalism-independent characterization. In *Proceedings of ACL workshop Reversible Grammar in Natural Language Processing*, Berkeley, 1991.

Hans Uszkoreit. Categorial unification grammar. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING)*, Bonn, 1986.

Kees van Deemter. Structured meanings. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

Kees van Deemter. *On the Composition of Meaning. Four variations on the Theme of Compositionality in Natural Language Processing.* PhD thesis, Universiteit van Amsterdam, 1991.

Jan van Eijck and Robert C. Moore. Semantic rules for english. In Hiyan Alshawi, editor, *The Core Language Engine*, pages 83–115. ACL-MIT press, 1992. chapter 5.

Gertjan van Noord, Joke Dorrepaal, Doug Arnold, Steven Krauwer, Louisa Sadler, and Louis des Tombe. An approach to sentence-level anaphora in machine translation. In *Fourth Conference of the European Chapter of the Association for Computational Linguistics*, Manchester, 1989.

Gertjan van Noord, Joke Dorrepaal, Pim van der Eijk, Maria Florenza, and Louis des Tombe. The MiMo2 research system. In *Proceedings of the Third International Conference on Theoretical and Methodological issues in Machine Translation of Natural Languages*, University of Texas at Austin, 1990.

Gertjan van Noord, Joke Dorrepaal, Pim van der Eijk, Maria Florenza, Herbert Ruessink, and Louis des Tombe. An overview of MiMo2. *Machine Translation*, 6:201–214, 1991.

Gertjan van Noord. BUG: A directed bottom-up generator for unification based formalisms. *Working Papers in Natural Language Processing, Katholieke Universiteit Leuven, Stichting Taaltechnologie Utrecht*, 4, 1989.

Gertjan van Noord. An overview of head-driven bottom-up generation. In Robert Dale, Chris Mellish, and Michael Zock, editors, *Current Research in Natural Language Generation.* Academic Press, 1990.

Gertjan van Noord. Reversible unification-based machine translation. In *Proceedings of the 13th International Conference on Computational Linguistics (COLING)*, Helsinki, 1990.

Gertjan van Noord. Head corner parsing. In Mike Rossner, C.J. Rupp, and Rod Johnson, editors, *Constraint Propagation, Linguistic Description and Computation*, Lugano, 1991. Working paper No. 5, Istitute Dalle Molle IDSIA.

Gertjan van Noord. Head corner parsing for discontinuous constituency. In *29th Annual Meeting of the Association for Computational Linguistics*, Berkeley, 1991.

K. Vijay-Shanker and Aravind K. Joshi. Feature structure based tree adjoining grammar. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

K. Vijay-Shanker, David J. Weir, and Aravind K. Joshi. Characterizing structural descriptions produced by various grammatical formalisms. In *25th Annual Meeting of the Association for Computational Linguistics*, Stanford, 1987.

K. Vijay-Shanker. Description theory, feature structures, and tree adjoining grammars, to appear.

Jürgen Wedekind, Jochen Dörre, Andreas Eisele, Jo Calder, and Mike Reape. A survey of linguistically motivated extensions to unification-based formalisms. Technical report, Dyana Esprit, Centre for Cognitive Science, University of Edinburgh, 1990. Dyana Deliverable R3.1.A.

Jürgen Wedekind. Generation as structure driven derivation. In *Proceedings of the 12th International Conference on Computational Linguistics (COLING)*, Budapest, 1988.

David J. Weir. *Characterizing Mildly Context-Sensitive Grammar Formalisms*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, 1988.

Pete Whitelock. Shake-and-bake translation. In Mike Rossner, C.J. Rupp, and Rod Johnson, editors, *Constraint Propagation, Linguistic Description and Computation*, Lugano, 1991. Working paper No. 5, Istitute Dalle Molle IDSIA.

Rémi Zajac. A transfer model using a typed feature structure rewriting system with inheritence. In *27th Annual Meeting of the Association for Computational Linguistics*, Vancouver, 1989.

Henk Zeevat, Ewan Klein, and Jo Calder. Unification categorial grammar. In Nicholas Haddock, Ewan Klein, and Glyn Morrill, editors, *Categorial Grammar, Unification Grammar and Parsing*. Centre for Cognitive Science, University of Edinburgh, 1987. Volume 1 of Working Papers in Cognitive Science.

Arnold M. Zwicky. Concatenation and liberation. In Anne M. Farley et al., editor, *Proceedings of the Twenty-Second Regional Meeting of the Chicago Linguistics Society*, Chicago, 1986.

Curriculum Vitae

Gertjan van Noord was born on 8 May 1961 in Culemborg. In 1979 he obtained the VWO certificate at the Bonifatius College in Utrecht. In 1980 he started to study at the Pedagogische Academie in 's Hertogenbosch. In 1983 he received the 'onderwijzer' degree. In 1983 he went to Utrecht to study Dutch Linguistics and Literature, and received the propedeuse in 1984 cum laude. In 1987 he graduated cum laude from the University of Utrecht in General Linguistics.

From 1 September 1987 until December 1990 he worked at the University of Utrecht as a researcher for the Eurotra project, located at the General Linguistics department. In 1991 he worked at the University of the Saar in Saarbrücken for the BiLD project at the Computational Linguistics department. Since 1 January 1992 he works as a lecturer at the Alfa-informatica department of the University of Groningen.

He is married with Petri Wijgergangs and has a son, Rik.

Gertjan van Noord
vakgroep Alfa-informatica RUG
PO Box 716
NL 9700 AS Groningen
+31-50-635935
`vannoord@let.rug.nl`

Acknowledgments

I would like to express my thanks to the members of the Utrecht MiMo2 group. I furthermore thank the members of the Eurotra group in Utrecht and of the other Eurotra groups, the members of the linguistics department in Utrecht, the students of my Prolog classes, the people at the Computational Linguistics department and the DFKI in Saarbrücken, and my new colleagues on the fourth and fifth floor of the Harmony building in Groningen.

During the development of the ideas presented in this thesis I received many valuable comments and encouragements from at least the following people: Hiyan Alshawi, Rolf Backofen, Sergio Balari, Gosse Bouma, Luis Damas, Joke Dorrepaal, Marc Dymetman, Jan van Eijck, Pim van der Eijk, Maria Florenza, Claire Gardent, Dale Gerdemann, Jack Hoeksema, Heleen Hoekstra, Pierre Isabelle, Mark Johnson, Martin Kay, Jan Landsbergen, Klaus Netter, Günter Neumann, Steve Pulman, Mike Reape, Herbert Ruessink, Stuart Shieber, Neil Simpkins, Gerd Smolka, Louis des Tombe, Hans Uszkoreit, Nino Varile, Susan Warwick, Jürgen Wedekind, Pete Whitelock.

Daarnaast gaat mijn speciale dank natuurlijk uit naar Anke, Gosse en Petri voor de hulp bij het voorbereiden van de promotiefestiviteiten.