Institute of
Computational Linguistics
University of Zurich

Link2Tree: A Dependency-Constituency Converter

Lizentiatsarbeit der Philosophischen Fakultät der Universität Zürich
Referent: Prof. Dr. Michael Hess

Stefan Höfler

shoefler@cl.unizh.ch

Zürich, April 2002

Stefan Höfler
Wiesenbachstrasse 7a
CH-9015 St. Gallen
+41 71 / 310 16 65
shoefler@cl.unizh.ch

Abstract

Link2Tree is a dependency-constituency converter written in PROLOG and developed for Link Grammar and ExtrAns. This thesis describes its architecture and functionality in detail. It demonstrates what constraints on link structures make them equivalent to constitent structures. This equivalence makes Link2Tree a deterministic converter, since a linkage corresponds to exactly one constituent tree. However, the linkages may need some preprocessing, which is called 'relinking' in Link2Tree, to ensure their equivalence to a particular form of constituent structure. The development and implementation of a conversion algorithm is described.

Link2Tree is a flexible program, which enables users to specify the form of X-bar theory they desire for the constituent structure delivered by the converter. Furthermore, users can freely choose the features they want to use in the constituent output. The possibilities of Link2Tree make it applicable to Government & Binding as well as PSG structures. If its rule set is tuned accordingly, Link2Tree is able to preserve all information that is stored in linkages for the constituent structures it returns.

The thesis is divided into three parts: Part I states the basic theoretical concepts of dependency, and link grammar in particular, as well as constituency. Part II describes the conversion algorithm as well as the architecture and functionality of the converter. Part III explains how users can develop their own rule set for the converter.

A note to the reader

This thesis comprises the theoretical background as well as a documentation of the Prolog program Link2Tree. If it is read from beginning to end, I strongly encourage the reader to have the program code at hand, especially when reading parts II and III. If this paper is used as a manual to Link2Tree, I recommend using the predicate index at the end of the thesis as a starting point for specific problems. If a particular predicate is not found in the predicate index, one may search for the predicate which calls it.

The implementer wanting to integrate Link2Tree into a larger system is referred to section 4 for the most basic program specifications, and to section 5 for a general overview of the program. A description of the installation and startup of Link2Tree is provided in appendix A.

Contents

1 Introduction

1.1 ExtrAns and Link Grammar

The subject of this thesis is situated in the ExtrAns research project at the University of Zurich. ExtrAns is a Prolog implementation of a practical Answer Extraction (AE) system. Answer Extraction systems retrieve phrases in textual documents to directly answer natural language questions.The domain of ExtrAns is the Unix manpages. Answer Extraction over technical manuals requires detailed linguistic analysis.

> [ExtrAns] uses full parsing, partial disambiguation, and anaphora resolution to generate minimal logical forms of the documents and the query. The search procedure uses a proof algorithm of the user query over the Horn clause representation of the minimal logical forms. Remaining ambiguities in the retrieved sentences are dealt with by graded highlighting.
>
> (Mollá et al., 2000b:1)

For the syntactic analysis of document sentences and user queries, ExtrAns uses the Link Grammar parser as presented (and developed) in Temperley et al. (2000). The Link Grammar (LG) system consists of a fairly fast parser and a dependency-related grammar of English. This particular implementation was chosen for ExtrAns because it is fast enough for the on-line processing of user queries and because it is able to handle unknown words and to skip unanalyzable parts of a sentence. For more information on the choice of the Link Grammar parser for ExtrAns, readers are referred to Mollá et al. (2000a,b).

The Link Grammar parser returns sets of labeled links which are called *linkages*. Each link describes a dependency relation between two words of the sentence. The link labels distinguish different types of relations (e.g. verb-object or noun-determiner). For later processing steps, ExtrAns adds a direction to every link. The reader is referred to section 2 for a more detailed description of the LG output and the notion of directionality in dependency grammar.

The disadvantage of using the Link Grammar parser in ExtrAns is, that it is strongly lexicalist and its output is Link Grammar-specific. A more generally used and broadly accepted representation of syntactic structure is preferable. Though the concepts of dependency can be applied to presumably any language, the Link Grammar system is specific to English. The current design of ExtrAns is based on the output returned by the Link Grammar parser and is therefore interspersed with Link Grammar-specific constructions and exceptions. This is detrimental to its understandability, the clarity of its design and its adaptability to new problems and tasks. In its current state it is not yet possible to replace the Link Grammar parser in ExtrAns with another parser. Thus, a separation of the actual ExtrAns system from the LG parser would be desirable.

1.2 Aim and scope

It is the goal of this thesis (and the Link2Tree project[1]) to supply a constituency-based interface for the Link Grammar parser in ExtrAns. With the help of such an interface the design of ExtrAns can be adapted to a more general representation of syntactic structure without being urged to dispense with the good results of the Link Grammar parser for the moment. A first attempt in this direction was made by the makers of the Link Grammar parser. For release 4.1 of the Link Grammar system, Temperley et al. (2000) created the Link Grammar Phrase Parser, which supplies a simplified constituency output along with the linkages.

This thesis discusses the conversion of Link Grammar linkages into constituency-based tree structures. The development of a conversion strategy and its implementation in Prolog (henceforth called Link2Tree) is commented on. Schneider (1998b:168ff.) may be seen as the starting point to this project:

> This subchapter [on converting linkages to constituency] would also merit a separate finals paper. In the short space given here I can only give some hints, without making practical texts or attempting an automatic conversion.
>
> (Schneider, 1998b:168)

The first part of this thesis briefly presents the concepts of dependency grammar, link grammar and constituency grammar, as far as they are relevant for this project. As for constituency grammar, features specific to Government & Binding theory (GB), Generalized Phrase Structure Grammar (GPSG) and Head-Driven Phrase Structure Grammar (HPSG) are discussed in particular. Nevertheless, this paper does not intend to be a comprehensive introduction to constituency grammar or one of its specific grammar theories respectively. Readers are referred to Haegeman (1994), Cook and Newson (1996), Radford

[1] I will speak of the 'Link2Tree project' when I mean the thesis and its practical component (i.e. the Prolog program Link2Tree) as a whole.

(1997) for an introduction to Government & Binding Theory, to Gazdar et al. (1985), Bennett (1995) for an introduction to GPSG and to Pollard and Sag (1994), Sag and Wasow (1999) for an introduction to HPSG.

Part II explains the general architecture of the Link2Tree parser and gives a detailed description of its algorithm, functionality and modules. The third part of the thesis provides the description of the syntax of the Link2Tree rule set and discusses a selection of individual link types in more detail.

The source code of the program as well as some instructions for its installation and startup are provided in the appendices at the end of the thesis.

Part I

Concepts

2 Link grammar and dependency grammar

It is the task of the Link2Tree project to find and implement an algorithm that converts the linkages returned by the Link Grammar Parser to constituent structures. To be able to do this it is necessary to have a general idea of the theoretical background of the Link Grammar Parser. The theoretical framework in which it is situated is link grammar. Link grammar is related to dependency-based theories of grammar as Schneider (1998b:1f.) points out right at the beginning of his thesis: "Link Grammar structures are – as the name suggests – links between the words of a sentence. Link grammar is very closely related to dependency grammars ...." This is also stated by the authors of Link Grammar:

> Meĺčuk's[2] definition of a dependency structure, and Gaifman's[3] proof that dependency grammar is context free imply that there is a very close relationship between these systems and link grammar. This is the case.
>
> (Sleator and Temperley, 1993:12)

Nevertheless there are also striking differences between dependency grammar and link grammar, which will be presented later. First, a brief survey of the basic concepts of dependency grammar is given before moving on to link grammar and the Link Grammar Parser in particular. As the theoretical linguistic background of the Link Grammar Parser is not the primary focus of this thesis, giving a concise introduction to either dependency theories of grammar or to the Link Grammar system is not intended. The features of link grammar and dependency are discussed as far as they are relevant for the conversion into constituency structure. Readers are referred to Fraser (1996), Weber (1997), Tarvainen (1981) for the former and to Sleator and Temperley (1993), Temperley et al. (2000) for the latter and to Schneider (1998b) for a detailed discussion of their interrelation as well as for a linguistic comparison of constituency, dependency and link grammar.

2.1 Dependency grammar

2.1.1 Binary dependency

Modern dependency grammar was introduced in Tesnière (1959). A first rule formalism was given to the theory by Hays (1964) and Gaifman (1965). The latter shows that

[2] Meĺčuk (1988)
[3] Gaifman (1965)

dependency grammar is context-free.[4] Bröker and Kruijff (1999) describe the general idea of dependency as well as its basic distinction from constituency:

> Dependencies are always relations between two parts of a larger whole, never between the whole and one of its parts. The latter relation is *constituency*. Since dependencies are considered basic in DG, constituency relations are secondary; they can always be inferred from the dependencies, and it is a matter of debate whether or not they need to be recognized formally.

Dependency grammars (DG) are to be seen as grammatical theories that assume syntactic structure to consist of (i) lexical nodes (representing words) and (ii) binary relations between them (*dependencies*).

2.1.2 Directionality: the notion of head

Bröker and Kruijff (1999) state that "[i]nformally and roughly put, a dependency-based perspective is constituted by distinguishing a head/dependent asymmetry, and describing the relations between a head and its dependents in terms of semantically motivated dependency relations." Dependency relations connect two words, of which one is the *governor* (or *head*) and the other is the *dependent*. The governor conditions the occurrence and/or the form of the dependent. Covington (1994) offers a concise description of the head/dependent relation in dependency grammar:

> The fundamental relation of DG is between head and dependent. One word (usually the main verb) is the head of the whole sentence; every other word depends on some head, and may itself be the head of any number of dependents. The rules of the grammar then specify what heads can take what dependents (for example, adjectives depend on nouns, not on verbs).
>
> (Covington, 1994:1)

It will be demonstrated later that the notion of head in dependency grammar is central in the developed conversion algorithm. A first hint for this idea can be found in Hudson (1990):

[4]Schneider (1998b:18) points out that "Järvinen and Tapanainen (1997:3) stress, however, that only the model used by Gaifman and Hays is context-free, and that this does not apply for all formalisms."

A further point of contact between dependency theory and modern syntactic theories [i.e. constituency-based theories] is the central place of the notion 'head' in X-bar theory – which indeed could be defined as a version of phrase structure grammar in which every phrase is required to have a lexical head, where 'head' is used in almost the same way as in dependency theory.

(Hudson, 1990:111f.)

Schneider (1998b:38-53) offers a detailed discussion of the notion of head in dependency as well as constituency theory. Readers are referred to his references for a more comprehensive insight into different argumentations on the notion of head in syntactic theories.

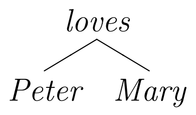
2.1.3 Dependency types

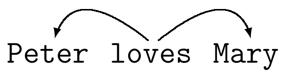
As it will be shown later, a third feature of dependency not discussed so far plays a role in the conversion of dependency structure into constituent structure: a dependency relation can be *typed* to distinguish different types of dependencies. Dependency types can contain morphologic, syntactic and semantic information. This is discussed in section 2.2.

2.1.4 Notation

Dependency grammar lacks phrasal nodes and since one word form depends on the other, individual words can act either as terminal or as non-terminal in a dependency-tree. This induces us to have a look at the notation of dependency structures. Sleator and Temperley (1993) describe dependency structure as defined by Melčuk (1988) as "a set of planar directed arcs among the words that form a tree. Each word (except the *root word*) has an arc out to exactly one other word, and no arc may pass over the root word." Apart from system-specific ways of notation like the Link Grammar linkages (see subsection 2.2) mainly two notational conventions are used to represent dependency structures. Tesnière (1959) favors the notation displayed in (2) to express dependencies, which is the so-called *stemma* notation. In recent publications, above all in those related to the Link Grammar system and to ExtrAns, the notation used in (3) is predominant. To avoid mixing up with notations of constituency – especially those of bare phrase structure (see section 3.7) – it will usually be declared, when the stemma notation is used.

(1) *Peter loves Mary.*

(2) 

(3) 

In (1), *Peter* and *Mary* both depend on *loves*. (2) is the stemma notation for (1). If the terminology of constituent trees is applied to the above dependency trees *Peter* and *Mary* act as terminals whereas *loves* is a non-terminal.

Dependency trees are usually construed as *projective*, i.e. without crossing branches. Nevertheless some instances of dependency grammar lift this limitation.

2.2 Link grammar

Link grammar (or 'Link Grammar' respectively, see below) is a formal grammatical system related to dependency grammar (and to categorial grammar, see Sleator and Temperley (1993:12f.)) developed by Temperley et al. (2000). Apart from the formalism, Temperley et al. (2000) have implemented a parser and a grammar for English using link grammar:

> We can roughly divide our work on link grammars [sic!] into three parts: the link grammar formalism and its properties, the construction of a wide-coverage link grammar for English, and efficient algorithms and techniques for parsing link grammars.
>
> (Sleator and Temperley, 1993:2)

In this thesis, lower-case is used when referring to the formalism in general and upper-case when referring to the Link Grammar system consisting of the Link Grammar Parser and a link grammar dictionary for English implemented and documented by Temperley et al. (2000). The reasons why the Link Grammar system was chosen for ExtrAns have already been commented on (section 1).

2.2.1 The link grammar formalism

In the abstract to their report Sleator and Temperley (1993) give a brief introduction to the basic idea of link grammar:

> We define a new formal grammatical system called a *link grammar*. A sequence of words is in the language of a link grammar if there is a way to draw *links* between

> words in such a way that (i) the local requirements of each word is satisfied, (ii) the links do not cross, and (iii) the words form a connected graph. [...] The formalism is lexical and makes no explicit use of constituents and categories.
>
> (Sleator and Temperley, 1993:1)

2.2.1.1 Lexicality The first point mentioned above gives evidence for the lexicality of link grammar: In link grammar each word is given a definition – a set of so-called *connectors* (see below) – describing how it can be used in a sentence, i.e. to what other words it can be linked. Therefore the actual grammar is distributed among the words. Such a system is said to be lexical. Sleator and Temperley (1993:3) suggest there are several advantages in a lexical system: (i) The grammar can easily be constructed incrementally and (ii) the grammar for irregularly behaving words can be expressed more easily since there is a separate definition for each word. Furthermore they state that (iii) lexicality allows the construction of useful probabilistic language models. Lexicality also implies that there is no explicit notion of constituents or categories in link grammar. Sleator and Temperley (1993:3) say that "constituents can be seen to emerge as contiguous connected collections of words attached to the rest of the sentence by a particular type of link." Though this is not the way they perceive link grammars, this perspective may give us some information about the conversion of link grammar structures to constituent structures.

2.2.1.2 Linking requirements As it has been shown so far, a link grammar consists of a set of words, each of which has a *linking requirement*. These linking requirements are expressed with *connectors* (i.e. a *formula* of connectors). Connectors can be seen as plugs (or sockets respectively) that are *satisfied* by a match with a compatible connector (an appropriate socket/plug). The word THE would be assigned a *D*+ connector, whereas WORM among others would be assigned a *D*−. THE thus requires a *D*− connector i.e. a *D* connector to its right, and WORM requires a *D*+ connector. The linking requirements of *the* and *worm* in (4) can thus be satisfied by matching their *D*+/*D*− connectors.

(4)
```
          D
     ┌──────────┐
     │          │
the[D+]  worm[D-]  ate the apple.
```

The reader is referred to Sleator and Temperley (1993:1f.) for a visualized explanation of the concept of connectors. Sleator and Temperley (1993:5) propose five meta-rules for a sentence to be of the language defined by the link grammar: (i) *satisfaction* (the links satisfy the linking requirements of each word in the sequence), (ii) *planarity* (links do not cross), (iii) *connectivity* (the links suffice to connect all the words of the sequence), (iv) *ordering* (when the connectors of a formula are traversed from left to right, the words they connect proceed from near to far) and (v) *exclusion* (no two links may connect the same pair of words).

2.2.2 Differences between link grammar and dependency grammar

It has been pointed out above that there is a strong relationship between link grammar and dependency grammar. There are, however, some important differences between the two theories. Schneider (1998b:125ff.) provides a detailed discussion of the comparison of link grammar and dependency grammar. His observations are followed in the rest of this subsection. He points out that there are five areas of difference between link grammar and classical dependency grammar: (i) labeled links, (ii) undirected links, (iii) word order and projectivity, (iv) root word, and (v) cycles.

2.2.2.1 Labeled links Link grammar links are – unlike in dependency grammar – labeled. Temperley et al. (2000) have developed a sophisticated labeling system for their link grammar of English. Schneider (1998b:125) remarks that "[t]his is not really a difference, as many variants of dependency grammar, perhaps the majority [...] also use labeled links." He furthermore points out that Covington (1994) introduces labeled links, indicating whether the dependent is *complement, adjunct* or *specifier* to its head. As it will be shown later, this will be crucial to our conversion of linkages to constituent, especially X-bar structures. Schneider (1998b:125) summarizes that "[i]n this sense, labeled links should facilitate a mapping from link structures to X-bar structures or to a functional or theta-role structure."

2.2.2.2 Undirected links One – if not *the* – major difference between link grammar and dependency grammar is that link grammar abandons the concept of directed links. Since its links are undirected, there is no specification of *head* and *dependent* in the original Link Grammar output. Schneider (1998b:127), however, points out that "it seems hardly possible to do semantics without a specification of direction...." This was probably one of the main reasons why directionality has been added to Link Grammar linkages in ExtrAns (see 2.3.2). The notion of head is crucial to the Link2Tree conversion algorithm of linkages to constituent structures, whereas Temperley et al. (2000) have developed a simple Link Grammar Phrase Parser disregarding this concept.

2.2.2.3 Word order and projectivity The +/- signs on connectors mark whether the word to be linked is expected to occur before (-) or after (+) the word in the lexical entry. Such rules do not occur in classical dependency grammar where word order is not relevant.

2.2.2.4 Root word Since the notion of head has been abandoned in link grammar, it is not self-evident to mark an element as the top head (*root word*) of a sequence. Most sentences, however, have a so-called *wall* link from an artificial word before the beginning of the sentence (the *wall*) usually to the subject. The idea of root word or root link respectively is discussed later in sections 8.2.2 and 12.3.1.

2.2.2.5 Cycles Since circularity in linkages is only introduced by the anaphoric binding link for the relative pronoun (B) it does not cause much problem to a conversion algorithm. Nevertheless, from a theoretical point of view, the possibility of circularity in link grammar is a fundamental difference to dependency grammar, where cycles are not allowed. Sleator and Temperley (1993) defend their approach for practical reasons:

> If we restrict ourselves to acyclic linkages, we run into another problem. This is that there is an exponential blow-up in the number of rules required to express the same grammar. This is because each disjunct of each word in the link grammar requires a separate rule in the dependency grammar.
>
> (Sleator and Temperley, 1993:12)

2.3 Explanation of the Link Grammar output

Taking a Link Grammar linkage as in (5) a few more concepts have to be commented on in addition to what has already been explained above. Generally speaking, the information contained in a Link Grammar linkage returned by ExtrAns is stored in the following elements: the *links*, the *direction* of the links, the link *types* and their *subscripts*, the *tokens* (including *walls*) and their *tags*. The linkage in (5) is what one receives if the Link Grammar parser is invoked from ExtrAns, using the predicates `start_parser/0`, `parse/2` and `print_links/1` (Mollá, 2000:49).[5] In (5) the directionality of the links can not be made out, therefore one has to work with the actual Prolog structure returned by `parse/2`, which is illustrated in (6).[6] The whole linkage is represented as a Prolog list containing two elements: a list of tokens and a list of links.

(5)
```
  +------------------Xp-------------------+
  |                +-------Op-------+     |
  +--Wd--+----Ss---+        +---Dmc-+     +-RW+
  |      |         |        |       |     |   |
///// cp[?].n copies.v042 the.d files.n2s . /////
```

(6)
```
[[[[/////, cp[?].n, copies.v042, the.d, files.n2s, ., /////],
[[0,5,h(nil),Xp], [0,1,h(l),Wd], [1,2,h(r),Ss], [2,4,h(l),Op],
[3,4,h(r),Dmc], [5,6,h(nil),RW]]]]]
```

2.3.1 Tokens, tags and walls

Tokens are listed in the first element of the Prolog linkage list. They are numbered in the order they appear in this list. The mark `[?]` after a token indicates that the token is not known in the lexicon. One main advantage of the Link Grammar parser is, that it is able to handle unknown words.

In ExtrAns, *tags* are added to the tokens. Schneider (1999) has developed and described a system of tags for different categories and word classes. For a detailed description of the singular tags, readers are thus referred to Schneider (1999).

Walls are dummy tokens at the beginning and end of every sentence. They are represented by `/////` in ExtrAns linkage structures. Walls take their own link types and have their own lexical entry. Because of the connectivity rule, it is necessary for the wall to be linked to the rest of the sentence in order for the sentence to be valid. As mentioned in 2.2.2.4, Link Grammar walls can be seen as a substitute for the root word of common dependency grammar.

2.3.2 Links and direction

The linking information of a Link Grammar linkage is stored in the second element of the linkage list. Every link in itself is a Prolog list consisting of four elements, the first two of which are the numbers of the linked tokens, the third indicating the direction and the fourth containing the type and subscripts of the link.

[5]Information on the ExtrAns system in this report refers to release 1.7 and to Link Grammar version 4.1.

[6]To increase readability empty spaces have been inserted after commas where necessary.

Direction is expressed in ExtrAns linkages in respect to the word order, or to the order of the tokens in the first element of the linkage list, respectively. The predicate `h/1` denotes the head of the link: `h(l)` indicates that the left token is the head, `h(r)` indicates that the right token is the head. If no head, i.e. no direction, is assigned to a link type, this is expressed by `h(nil)`. It has already been pointed out that there is no directionality in original link grammar links. Directionality thus was added by the creators of ExtrAns. The directions for the singular link types are defined by the predicate `add_dep_info/5` in the file `link_grammar.pl` of the ExtrAns source directory.

2.3.3 Link types and subscripts

Temperley et al. (2000) have developed an elaborate system of link types for English. It is the advantage of this system that it can deal with many idiomatic constructions of English and thus has a wide coverage. It is this feature of the Link Grammar system, however, that makes it rather language-specific. Temperley et al. (2000) provide an extensive description of their link types. Readers are referred to the former description for detailed information on specific link types.

Apart from link types, which are equivalent to the connectors described above (2.2.1.2), Temperley et al. (2000) make use of yet another system to differentiate constructions and include additional information in their analysis. Subscripts are used along with link types. They are lower-case letters (connectors or link types, respectively, consist of one or more upper-case letters) following the connector name. Depending on the link type they specify, subscripts can provide morphologic, syntactic or semantic information.

3 Constituency grammar

Probably the most influential branch of constituency grammar has been evolved by Noam Chomsky. In Chomsky (1957), he introduces an approach to syntax known as *Transformational Grammar* (TG). Since then, TG has constantly been further developed. The most prominent theory in this tradition is Chomsky's *Government & Binding* theory (GB), which was first presented in Chomsky (1981) and revised in Chomsky (1986a). The most recent development is Chomsky's so-called *Minimalist Program* (Chomsky, 1995), which abandons many of the concepts of the earlier theories.

A major alternative to the theories of the Chomskian tradition is *Phrase Structure Grammar*. In fact, Phrase Structure Grammar consists of two different but related approaches: *Generalized Phrase Structure Grammar* (GPSG) and *Head-driven Phrase Structure Grammar* (HPSG). GPSG was developed and presented in Gazdar et al. (1985) and HPSG in Pollard and Sag (1987, 1994). Note that the term 'Phrase Structure Grammar' can be either used in the above sense denoting the branch of constituency grammar represented by GPSG and HPSG, or it can be used as a synonym for constituency grammar as a whole. In this thesis, it is used in the former sense.

This section is intended to give a brief survey over the concepts shared by all theories of constituency grammar rather than a detailed description of the features of one particular theory.

3.1 Constituency

Bröker and Kruijff (1999) have been quoted as they state the most basic difference between dependency and constituency in general: "Dependencies are always relations between two parts of a larger whole, never between a whole and one of its parts. The latter relation is called *constituency*." It can thus be said that the basic notion of constituency is, that two (or more) units compose a larger unit. Constituency grammar assumes about sentence structure "that it is organized hierarchically into 'phrases' (hence 'phrase structure'), and that grammatical relations such as 'subject' and 'object' are redundant." (Sag and Wasow, 1999:421). This distinctively American contribution to the theory of syntax analysis was suggested by Bloomfield (1933).

Bloomfield suggested that sentences should be analyzed by a process of segmentation and classification: segment the sentence into its main parts, classify these parts, then repeat the process for each part, and so on....

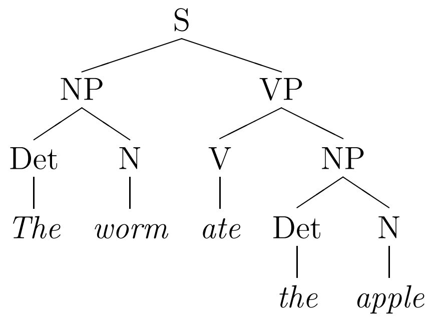
(Sag and Wasow, 1999:421)

Borsley (1997:41-49) and Burton-Roberts (1997:15-18) provide a set of tests to check whether a particular part of sentence is a proper constituent or not.

3.2 Tree diagrams

Constituent structures are usually represented by either tree diagrams (7) or labeled brackets (8). Both notational systems are equivalent.

(7)



(8) $[_S[_{NP}[_{Det}\textit{The}][_N\textit{worm}]][_{VP}[_V\textit{ate}][_{NP}[_{Det}\textit{the}][_N\textit{apple}]]]]$

In a tree diagram, a sequence of elements is represented as a constituent if there is a node that dominates all these elements and no others. A node X is the ‘mother’ of two other nodes Y and Z, if either Y and Z is immediately dominated by X. Y and Z in turn are called ‘daughters’ of X and ‘sisters’ to each other. Nodes are usually labeled with ‘categories’ (see below). Nodes bearing lexical categories, such as e.g. Det, N and V, are called ‘terminal nodes’. The term ‘leaf’ refers to the word attached to a terminal node in a tree diagram.

3.3 Feature structures

In (7) above, nodes are labeled with grammatical categories. Such labels stand for complexes of properties.

Instead of associating words in the lexicon with a single atomic category, a lexical category can be treated as a complex of grammatical properties. To model such complexes, we use the notion standardly referred to as *feature structure.*

(Sag and Wasow, 1999:48)

Linguistic properties are commonly expressed by feature-value pairs. Several feature-value pairs of an entity can be combined to a feature structure. Bennett (1995:39) uses the notation given in (9) for such feature structures. (9) denotes the properties of a plural noun such as *worms*. The +/− notation is used for features that take Boolean values.

(9) {[+N], [BAR 0], [+PLU]}

Another common way of representing feature structures are so-called 'attribute-value matrices' (AVM).[7] This is the notation that is most commonly used in HPSG. (10) illustrates the general pattern of such attribute-value matrices. (At the bottom left corner of the matrix, the so-called 'type' is indicated. This is not relevant for our purposes.[8])

(10)
$$ {}_{\textit{type}}\begin{bmatrix} \text{FEATURE}_1 & \text{value}_1 \\ \text{FEATURE}_2 & \text{value}_2 \\ \ldots & \\ \text{FEATURE}_n & \text{value}_n \end{bmatrix} $$

3.4 Complex feature structures

Phrase structure grammar provides a method to express agreement in grammar rules. An index (variable) linking two feature values forces two distinct nodes in a tree admitted by a rule to have identical values for the given feature. (11) gives a simple rule for agreement of number and person between the noun and its determiner in an NP. Tags as in (11) can only occur in the unresolved feature structures (feature structure descriptions) of a grammar rule. Unresolved feature structures have to be distinguished from resolved feature structures, which are the result of parsing a particular sequence of words, where variables are instantiated.

[7]The terms 'feature structure', 'attribute-value matrix', 'feature matrix' can be used synonymously.

[8]Sag and Wasow (1999) define the term 'type' in the glossary of their book:

> Elements of any collection can be sorted into types, based on similarities of properties. [...] Particular features are appropriate only to certain types of entities, and constraints on possible feature-value pairing are also associated with some types. The types of linguistic entities are arranged in an inheritance hierarchy. The type hierarchy is especially important for capturing regularities in the lexicon.
>
> (Sag and Wasow, 1999:445f.)

(11)
$$ {}_{\textit{phrase}}\begin{bmatrix}\text{POS} & \text{N}\\ \text{NUM} & \boxed{1}\\ \text{PER} & \boxed{2}\end{bmatrix} \rightarrow {}_{\textit{word}}\begin{bmatrix}\text{POS} & \text{Det}\\ \text{NUM} & \boxed{1}\\ \text{PER} & \boxed{2}\end{bmatrix} {}_{\textit{word}}\begin{bmatrix}\text{POS} & \text{N}\\ \text{NUM} & \boxed{1}\\ \text{PER} & \boxed{2}\end{bmatrix} $$

The notation of (11) can be simplified by introducing complex values for features, that is, feature structures within feature structures as illustrated in (12).

(12)
$$ {}_{\textit{phrase}}\begin{bmatrix}\text{POS} & \text{N}\\ \text{AGR} & \boxed{1}\end{bmatrix} \rightarrow {}_{\textit{word}}\begin{bmatrix}\text{POS} & \text{Det}\\ \text{AGR} & \boxed{1}\end{bmatrix} {}_{\textit{word}}\begin{bmatrix}\text{POS} & \text{N}\\ \text{AGR} & \boxed{1}\begin{bmatrix}\text{NUM}\\ \text{PER}\end{bmatrix}\end{bmatrix} $$

3.5 The Head Feature Principle

Sag and Wasow (1999) describe the notion of 'head' in constituency grammar:

> [T]he phrases of human language usually share certain key properties (nounhood, verbhood, prepositionhood, etc.) with a particular daughter within them – their *head*.
>
> (Sag and Wasow, 1999:47)

They further point out that "[t]he term is used ambiguously to refer to the word that functions as head of the phrase and any subphrase containing that word." (Sag and Wasow, 1999:436).

Sag and Wasow (1999) define a feature HEAD for every constituent, containing the so-called 'head features', and say that "in any headed phrase, the HEAD value of the mother and the HEAD value of the head daughter must be unified." (Sag and Wasow, 1999:63). In HPSG, this is called the *Head Feature Principle* and expressed in rule (13), where the H in front of the feature structure denotes the head daughter.

(13)
$$ {}_{\textit{phrase}}\begin{bmatrix}\text{HEAD} & \boxed{1}\end{bmatrix} \rightarrow \text{H}\begin{bmatrix}\text{HEAD} & \boxed{1}\end{bmatrix} \ldots $$

By definition, HEAD will always take the part of speech of the constituent as one of its values.

The idea of 'head features' is also common to GPSG, although they are not combined into one feature with a complex value. The principle stated above is called *Head Feature Convention* (HFC) in GPSG: "A node and its head must share the same head features."

(Bennett, 1995:55). In addition to this, GPSG also knows the notion of ‘foot features’, i.e. features shared by the mother node and one of its daughters. GPSG defines the so-called *Foot Feature Principle* (FFP), which says that “any foot feature instantiated on a daughter in a local tree must also be instantiated on the mother in that tree and *vice versa*.” (Bennett, 1995:105).

3.6 X-bar theory

A model of syntax that makes extensive use of the idea of ‘head’ is X-bar theory. X-bar theory is applied widely within Government & Binding theory, as well as GPSG. One of the guiding principles of X-bar theory is, that phrases are to be seen as ‘projections’ of their lexical head. This means that the category (part of speech) of the phrase matches that of its head. As it was shown above, HPSG defines the same principle as it states that the complex value of the feature HEAD always takes the part of speech as one of its values. Bennett (1995) comments on the use of heads in X-bar theory:

> This emphasis on heads makes X-bar theory closer to *dependency grammar*, but it remains different in that it still makes use of phrases and so of constituents larger than a word.
>
> (Bennett, 1995:15)

“[X-bar theory] insists that phrases must be ‘endocentric’: a phrase always contains at least a head as well as other possible constituents.” (Cook and Newson, 1996:135). X-bar theory introduces X-bar levels to distinguish different kinds of projections. In GPSG, this is represented as a feature BAR taking the values 0–2.[9] X-bar theory states principles for every level of projection. Cook and Newson (1996:147) summarize these principles in (14–17).

(14) A phrase always contains a head of the same type.

(15) A two-bar category consists of a head that is a single-bar, a specifier position, and a possible specifier:
X” → specifier X’

[9]X” is alternatively represented as $\overline{\overline{X}}$, XP or X2, X’ as $\overline{X}$ or X1, and X as X0.

(16) A single-bar category contains a head with no bars and possible complements[10]:
X' → X complement(s)

(17) A single-bar category can also contain a further single-bar category and an adjunct:
X' → X' adjunct

Bennett (1995:15) points out that "GPSG adopts a rather unrestrictive version" of X-bar theory and represents its principles in formula (18), where the first value of each tuple indicates the category and the second value of the tuple indicates the bar level. The value MAX stands for 'maximal projection', i.e. bar level 2. (18) is the simplified notation commonly used in GPSG.

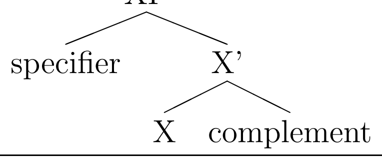
(18) [X,n] → [Y, max]* [X, m] [Y, max]*
(where m = n or m = n–1)

Chomsky (1986b:81) states on the notion of 'complement' that "phrases typically consist of a head [...] and an array of complements determined by the lexical properties of the head." One says that a lexical item 'subcategorizes' for its complement(s). The transitive verb *love*, for instance, subcategorizes for an NP-complement. In HPSG, this is expressed within the feature structure of the lexical item as illustrated in (19). (20) is the common GPSG notation for the same feature structure.

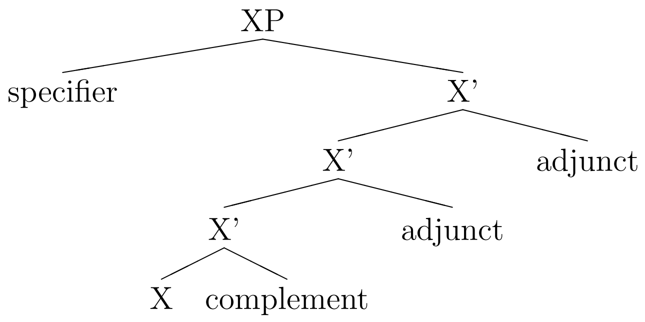
(19) $\left\langle \textit{love}, \begin{bmatrix} \text{HEAD} & \text{V} \\ \text{COMPS} & \text{<NP>} \end{bmatrix} \right\rangle$

(20) V [SUBCAT <NP>]

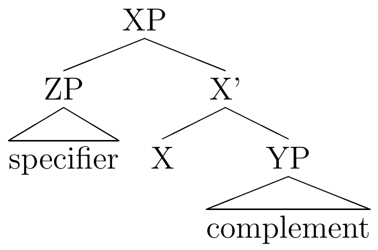
Principles (15–17) mean that in X-bar theory the minimal structure of a phrase is the one presented in (21). Any number of adjuncts can be added to this basic tree, each time inserting another single-bar level as illustrated in (22).

(21)


[10]Double objects, however, may occur in English (see also 8.1.3).

(22)


The occurrence of complements in an X-bar constituent structure depends on the subcategorizing of the lexical head. Specifiers, however, always open a slot, the specifier position, even if there is no specifier overtly present. In GB, empty specifier positions serve as landing sites for moved constituents. Movement is briefly discussed in section 3.8. Adjuncts are optional. Readers interested in tests for distinguishing complements from adjuncts are referred to Matthews (1981:ch. 6) and Somers (1984). Note that specifiers, complements and adjuncts are themselves maximal projections, i.e. two-bar categories (23).

(23)


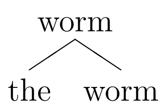
GPSG, however, claims that certain word-classes (e.g. Det) do not project to higher-level phrases. Bennett (1995:18) uses the term 'minor categories' for such classes and expands (18) to (24).

(24) [X,n] → {[Y, max]|minor}* [X, m] {[Y, max]|minor}*
(where m = n or m = n–1)

3.7 Bare phrase structure

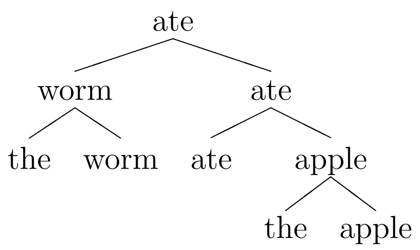
In his recent work, the Minimalist Program, Chomsky abandons X-bar theory as a fundamental principle. In Minimalist grammar, the basic operation to build trees is *Merge*. Merge combines two elements, words or phrases, and selects one – the head – , which provides the properties for the combined set and is taken as its label. The sequence *the worm* is merged to (25) in Minimalist grammar. This form of representation is called 'Bare Phrase Structure'.

(25)



Note that using ‘worm’ as the category label does not imply that this constituent is to be represented by the word *worm*. The elements of tree (25) are feature structures which are, for simplicity, represented by the orthographic form of their lexical head. In Minimalist grammar, the sentence *The worm ate the apple* is represented as a Bare Phrase Structure tree like (26).

(26)



Nevertheless, X-bar theory can still be derived from this structure: Maximal projections are simply the furthest an element projects, single-bar categories are those projections which are neither words nor maximal. Therefore, complements, specifiers and adjuncts are definied as well, as they are defined structurally rather than functionally in constituency grammar, anyway. Complements are sisters to lexical items, specifiers are non-head daughters of maximal projections – both demanded by subcategorization of the head.

Note that in the recent, commonly accepted DP-hypothesis *the* would have been taken as the head of *the worm*, forming a determiner phrase (DP).

Obviously, Bare Phrase Structure is reminiscent of dependency structures. Schneider (1998b:69–75) thoroughly describes the relation between the Minimalist Program and dependency grammar. He states that, as a derived concept, constituents also exist in dependency, in that “[a] dependency constituent consists of a head and all its dependents.” (Schneider, 1998b:168).

3.8 Syntactic representations

One basic difference between the theories of the Chomskian tradition on the one side and GPSG and HPSG on the other side is the number of levels of syntactic representation. Whereas GPSG and HPSG assign just one single syntactic representation to a sentence and are therefore labeled ‘monostratal’ theories, TG and GB assume that sentences have two levels of syntactic representation: ‘D-structure’ (deep-structure) and ‘S-structure’ (surface-structure).

D-structure

This level encodes the lexical properties of the constituents of the sentence. It represents the basic argument relations in the sentence.

S-structure

This level reflects the more superficial properties of the sentence; the actual ordering of the elements in the surface string, and their case forms.

(Haegeman, 1994:304f.)

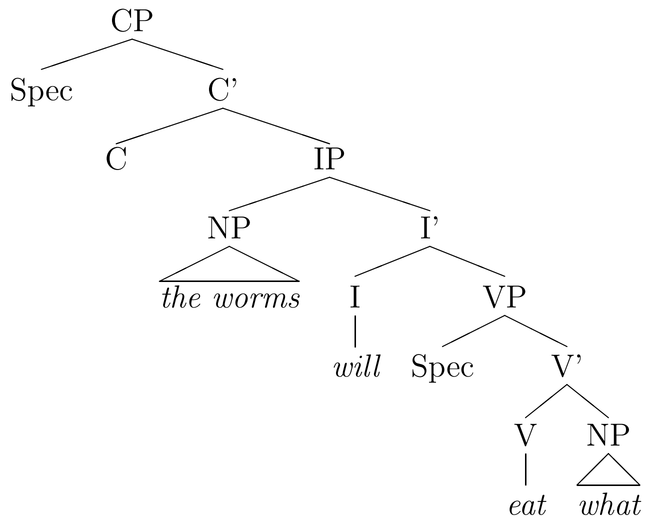
The two levels of representation are interrelated with each other by means of movement transformations. D-structure represents the underlying form of the sentence before movement. S-structure describes the related form of the sentence after movement: elements which originate in some positions at D-structure may have moved elsewhere at S-structure. In order not to loose syntactic and semantic information, however, the original positions of the moved elements are indicated at S-structure. The places from which elements have moved are marked by 'traces', symbolized as *t*.

Movement implies that appropriate 'landing sites' for the elements to be moved are provided in D-structure. Therefore, empty constituents play an important role in GB. The example in (27–29) may illustrate two types of movement: (i) 'head-movement', which is the movement of auxiliaries from I (inflection) to C (complementizer), and (ii) 'wh-movement', the movement of *wh*-constituents to the specifier of CP. Both types of movement imply the existence of functional categories such as I and C, who may not be overt in a sentence. (28) gives the D-structure for (27), (29) is its S-structure including the traces of the moved constituents.[11]

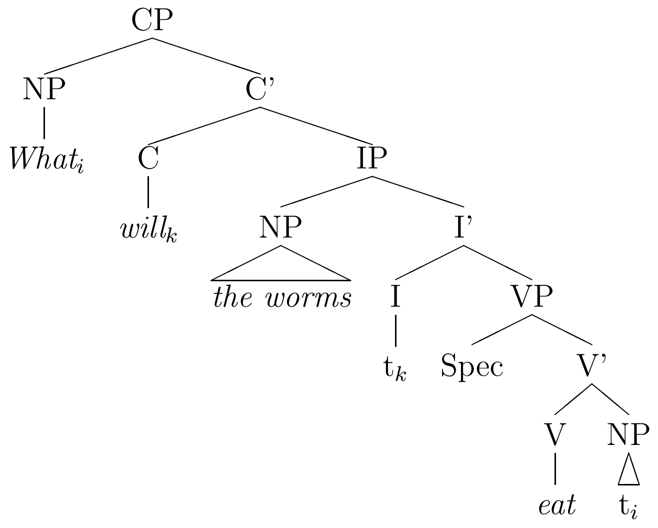
(27) $What_i$ $will_k$ *the worms* t_k *eat* t_i*?*

[11]Recent approaches would additionally state that the subject has moved from inside the specifier of VP to its position in IP. This is called the *VP-internal subject hypothesis.* For evidence that subjects originate in the specifier of the VP, readers are referred to Radford (1997:318ff.).

(28)



(29)

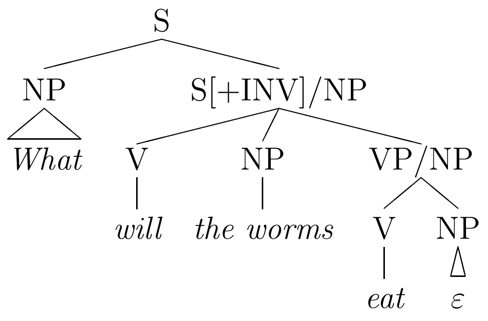


The third movement type defined in GB, which is not illustrated in (27–29) above, is 'NP-movement', in which an NP is moved to an empty subject position. NP-movement is associated with passive constructions.

D-structure is especially interesting for our purposes as it is much closer to semantics than any other level of syntactic representation.

GPSG and HPSG, however, are monostratal theories, that is, they give only one level of syntactic representation for a sentence, namely what would be called a surface structure in GB. The problems faced by movement in TG/GB are solved by defining extra features in GPSG and HPSG. In GPSG, for instance, the feature SLASH – represented as /XP – treats what is called *wh*-movement in GB, determining a required sister-constituent. The feature INV is introduced to handle subject-auxiliary inversion; see (30).

(30)



Part II

The converter

4 Program specifications

4.1 Program requirements

In this part of the thesis, the Link2Tree converter is presented, commented and discussed. Link2Tree is a Prolog program developed to convert Link Grammar linkages to constituent structures for ExtrAns. Readers are referred to section 1 for a discussion of its prerequisites. The program is written in Prolog, applying the conventions of SICStus Prolog 3.8. ISO-Prolog standards, however, are followed throughout Link2Tree. The program is designed to work in interaction with ExtrAns 1.7 and Link Grammar 4.1 as described in section 4.3 and appendix A.

4.2 Input, internal representation and output

Link2Tree takes a Link Grammar linkage as its input and returns a constituency structure. Rules for this process are specified in the module `linkfeatures.pl` (see part III). (32) is the graphical display of the linkage which ExtrAns and Link Grammar return for sample sentence (31). Link2Tree takes the structure of (32) in its Prolog form as its input and returns the constituency structure graphically displayed in (33).[12]

(31) *cp copies the files.*

(32)
```
  +-------------------Xp------------------+
  |                +-------Op-------+     |
  +--Wd--+----Ss---+        +---Dmc-+     +-RW+
  |      |         |        |       |     |   |
///// cp[?].n copies.v042 the.d files.n2s . /////
```

[12]It is evident from the empty functional categories D and I that the rule set in `linkfeature.pl` has been defined in a Government & Binding-like style for this sample.

(33)

```
                       i2
                       |
     +----------------------+
     d2                     i1
     |                      |
     |         +----------------+
     d1        i0               v2
     |         |                |
+-------+      |                |
d0      n2     |                v1
|       |      |                |
|       |      |       +-------------+
|       n1     |       v0           d2
|       |      |       |             |
|       |      |       |             |
|       n0     |       |             d1
|       |      |       |             |
|       |      |       |        +-------+
|       |      |       |        d0      n2
|       |      |       |        |       |
|       |      |       |        |       |
|       |      |       |        |       n1
|       |      |       |        |       |
|       |      |       |        |       |
|       |      |       |        |       n0
|       |      |       |        |       |
|       |      |       |        |       |
[]    cp[?]    []    copies    the    files
```

(32) and (33), however, are mere graphical representations for the underlying Prolog structures. The actual ExtrAns output that is transferred to Link2Tree is the Prolog list in (34). The graphical display in (33) is built upon the underlying Prolog structures given in (35–36). (35) is a Prolog list defining the tree structure, where node numbers refer to the corresponding feature structures. (36) is a set of dynamically asserted terms of the form `feature(NodeNr, Featue=Value)`, which defines the feature structure for each node in the tree. As you can see from the extracts in (36), the `feature/2` Prolog facts are asserted in the module `features.pl`.

(34)
```
[[[['/////', 'cp[?].n', 'copies.v042', 'the.d', 'files.n2s', '.', '/////'],
[[0,5,h(nil),'Xp'], [0,1,h(l),'Wd'], [1,2,h(r),'Ss'], [2,4,h(l),'Op'],
[3,4,h(r),Dmc], [5,6,h(nil),RW]]]]]
```

(35)
```
[9,[10,[11,[8,[]], [12,[13,[1,'cp[?].n']]]]],[14,[7,[]],[15,[16,
[2,'copies.v042'],[17,[18,[3,'the.d'],[19,[20,[4,'files.n2s']]]]]]]]]
```

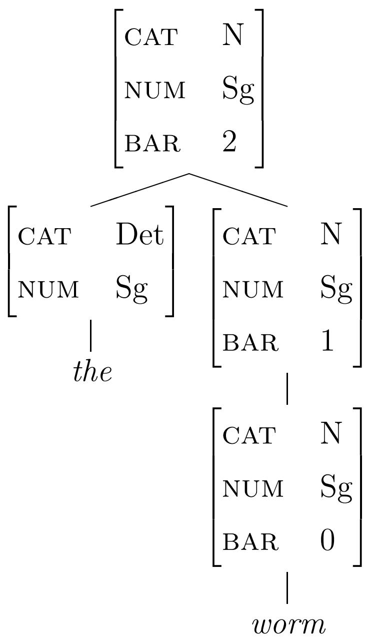
(36)
```
...
features:feature(9, cat=i).
features:feature(9, num=sg).
features:feature(9, bar=2).
features:feature(10, position=0.9).
features:feature(10, cat=d).
features:feature(10, num=sg).
features:feature(10, bar=2).
...
```

To illustrate the structure of the Link2Tree output in a simpler manner, it can be stated that the Prolog list in (37) and the Prolog facts in (38) correspond to the HPSG-like tree in (39). Note that the features POSITION and HD, which denote the position of a constituent in the sentence and the lexical head of a constituent respectively, are not shown explicitly in (39). Mother nodes have inherited these two features from their head daughters as they are defined as head features in the rule set.

(37)
```
[4,[1,'the'],[3,[2,'worm']]]
```

(38)
```
feature(1, position=0).
feature(1, hd=the).
feature(1, cat=det).
feature(1, num=sg).
feature(2, position=1).
feature(2, hd=worm).
feature(2, cat=n).
feature(2, num=sg).
feature(2, bar=0).
feature(3, position=1).
feature(3, hd=worm).
feature(3, cat=n).
feature(3, num=sg).
feature(3, bar=1).
feature(4, position=1).
feature(4, hd=worm).
feature(4, cat=n).
feature(4, num=sg).
feature(4, bar=2).
```

(39)

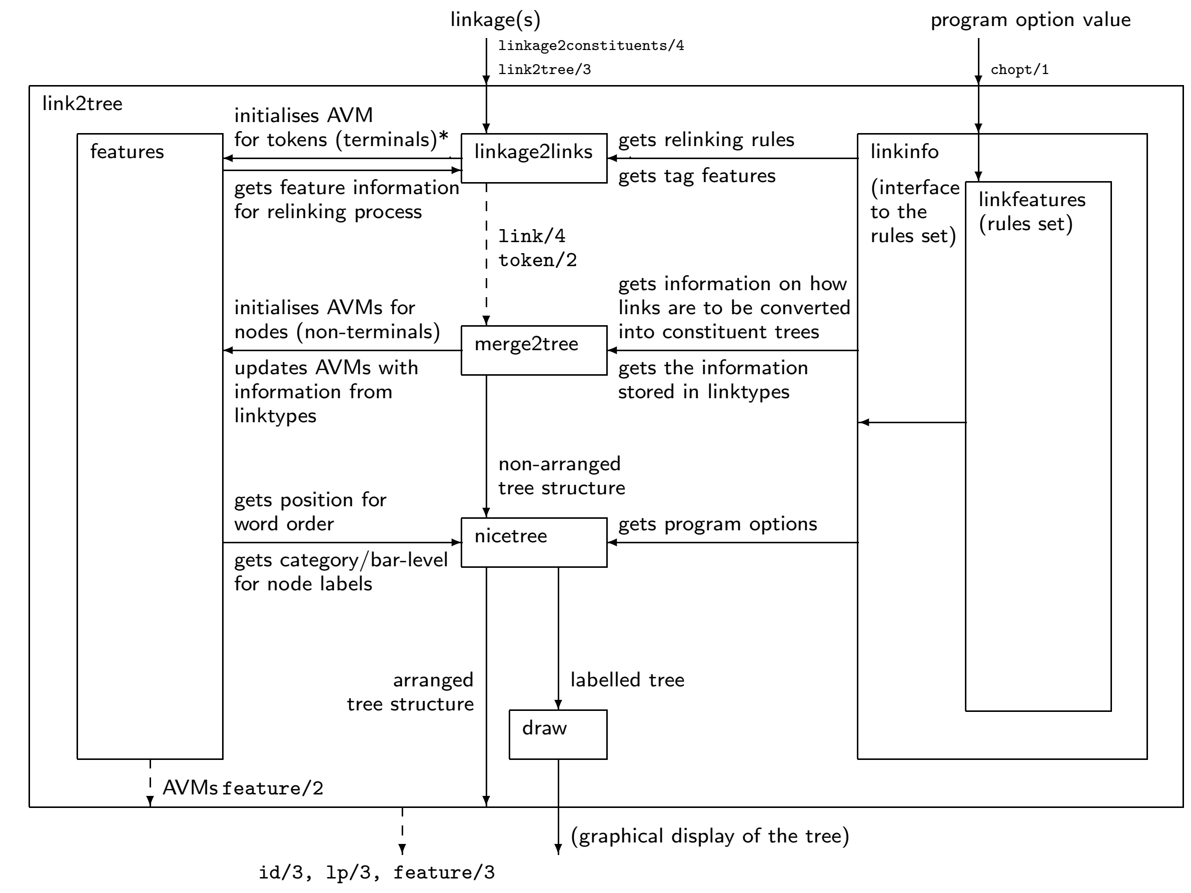


This internal representation of tree structures, however, is post-processed before returned. The final output of Link2Tree consists of three predicates: `id/3` to express immediate dominance relations, `lp/3` for linear precedence and `feature/3` to represent the feature structures of the nodes. The first argument of each predicate is the number of the parsed sentence. The three output predicates can be generally described as `id(SentenceNr, MotherNodeID, DaughterNodeID)`, `lp(SentenceNr, LeftNodeID, RightNodeID)` and `feature(SentenceNr, NodeID, Feature=Value)`. Note that `lp/3` only denotes relations of *immediate* linear precedence. (40) is the above tree structure in its final output form.

(40)
```
id(0, 4, 1).
id(0, 4, 3).
id(0, 3, 2).
lp(0, 1, 3).
feature(0, 1, position=0).
feature(0, 1, hd=the).
feature(0, 1, cat=det).
feature(0, 1, num=sg).
feature(0, 2, position=1).
feature(0, 2, hd=worm).
feature(0, 2, cat=n).
feature(0, 2, num=sg).
feature(0, 2, bar=0).
feature(0, 3, position=1).
feature(0, 3, hd=worm).
feature(0, 3, cat=n).
feature(0, 3, num=sg).
feature(0, 3, bar=1).
feature(0, 4, position=1).
feature(0, 4, hd=worm).
feature(0, 4, cat=n).
feature(0, 4, num=sg).
feature(0, 4, bar=2).
```

4.3 The interface to other programs

If Link2Tree is to be integrated into ExtrAns, the predicate `linkage2constituents/4` is to be used as an interface. The first three arguments define the input of the predicate. In the first argument, the number of the parsed sentence is to be provided. This number is delivered to the first argument of the three output predicates `id/3`, `lp/3` and `feature/3`. The second argument takes the set of Link Grammar linkages returned by the ExtrAns predicate `parse/2`. In the third argument, one must specify which linkage shall be converted. It therefore takes the number of the result linkage to be processed by Link2Tree. Finally, the fourth argument of `linkage2constituents/4` returns the resulting constituent structure in its Prolog list form, like in (35) above. This argument, however, may usually not be necessary since the resulting constituent structure can be accessed through the predicates `id/3`, `lp/3` and `feature/3`, which have to be imported from Link2Tree (i.e. from its module `link2tree.pl`).


linkage(s)
linkage2constituents/4
link2tree/3
program option value
chopt/1
link2tree
features
initialises AVM for tokens (terminals)*
linkage2links
gets relinking rules
gets tag features
linkinfo
(interface to the rules set)
linkfeatures (rules set)
gets feature information for relinking process
link/4
token/2
gets information on how links are to be converted into constituent trees
initialises AVMs for nodes (non-terminals)
merge2tree
updates AVMs with information from linktypes
gets the information stored in linktypes
non-arranged tree structure
gets position for word order
nicetree
gets program options
gets category/bar-level for node labels
arranged tree structure
labelled tree
draw
AVMs feature/2
(graphical display of the tree)
id/3, lp/3, feature/3

* position (word order) and information from the tag

Figure 1: General architecture of Link2Tree

5 General architecture

Link2Tree is implemented as a module for ExtrAns. It involves itself several modules written in Prolog. It implies several modules for the actual converter plus one rule set file as well as the two startup scripts discussed in appendix A and the tree drawing module `draw.pl` by Mark Holcomb.

Figure 1 shows the general architecture of the program: Prolog predicates that serve as program interfaces are printed in tiny letters; solid lines mean data flow in the direction of the arrow; dashed lines denote dynamically asserted facts to be imported; boxes stand for modules.

5.1 A brief survey of the modules

5.1.1 `link2tree.pl`

The module `link2tree.pl` provides the main routine(s). It contains the interfaces for the user (`link2tree/3`) and other programs (`linkage2constituents/4`). While processing a linkage input it calls the modules `linkage2links.pl, merge2tree.pl` and `nicetree.pl`, as well as Mark Holcomb's module `draw.pl` if the result is to be displayed on the screen. It also does the post-processing, which turns the internal representation of the resulting constituent tree into sets of `id/3`, `lp/3` and `feature/3` facts.

5.1.2 `linkage2links.pl`

The module `linkage2links.pl` reformats the linkage input, which is handed over to it from `link2tree.pl`. It converts the complex Prolog list representing the linkage into Prolog terms for links (`link/4`) and tokens (`token/2`). These are stored as dynamically asserted Prolog facts, and a feature structure is initialized for each token, using the corresponding tools provided by the module `features.pl`.

Furthermore, `linkage2links.pl` is responsible for the so-called relinking process. That is, it changes the structure of the linkage according to relinking rules defined in the rule set (`linkfeatures.pl`) to allow a convenient conversion into constituent structures. This enables the implementer to correct unfavourable Link Grammar links.

5.1.3 `merge2tree.pl`

The module `merge2tree.pl` is the actual converter. It does the conversion of the link structure stored in `linkage2links.pl` into a constituent structure. During this process, it makes use of the facilities provided by the module `features.pl` to initialize feature structures for nodes or to update those of the tokens respectively. The output of `merge2tree.pl` is a non-arranged constituent tree, the nodes of which are the identifiers of feature structures stored in `features.pl`. This tree represents dominance only but not necessarily linear precedence (word order).

5.1.4 `features.pl`

The module `features.pl` does the feature handling. It provides tools to build and update feature structures. It stores feature structures as dynamically asserted Prolog facts (`feature/2`).

5.1.5 `nicetree.pl`

The module `nicetree.pl` has two functions: (i) it rearranges the constituents of the tree structure returned by `merge2tree.pl` according to the word order of the original sentence, and (ii) it provides a tool to replace feature structure identifiers in the nodes of the tree by node labels like N0, N1, N2 (or N, N bar, NP). The first function is accessed through `wordorder/2`, the second through `nodelabels/2`.

5.1.6 `linkinfo.pl`

The module `linkinfo.pl` is the interface to the declarative rule set in `linkfeatures.pl`. It provides tools to handle and apply the declarations made in the rule set. There is no direct access to `linkfeatures.pl` from any module other than `linkinfo.pl`.

5.1.7 `linkfeatures.pl`

The module `linkfeatures.pl` is the declarative database of the program. Its rule set contains the information how links must be converted into constituents. Furthermore, the program options are stored in this module. Linguistically motivated alterations or adaptations need only be made in this file.

If `merge2tree.pl` was said to be the "parser-part" of the program, `linkfeatures.pl` would be its "grammar-part".

5.2 A step by step example

The following example shall illustrate step by step how a given linkage is transformed into a corresponding constituent structure.

5.2.1 Input

User query (41) makes ExtrAns and Link Grammar to return the linkage output in (42). (43) represents the underlying prolog structure of (42), which is handed over to `link2tree.pl` and from there to the module `linkage2links.pl`.

(41)
```
| ?- link2tree("cp copies the files.", 0, Tree).
```

(42)
```
    +------------------Xp-----------------+
    |               +-------Op-------+     |
    +--Wd--+----Ss---+        +---Dmc-+     +-RW+
    |      |         |        |       |     |   |
///// cp[?].n copies.v042 the.d files.n2s . /////
```

(43)
```
[[[[/////, cp[?].n, copies.v042, the.d, files.n2s, ., /////],
[[0,5,h(nil),Xp], [0,1,h(l),Wd], [1,2,h(r),Ss], [2,4,h(l),Op],
[3,4,h(r),Dmc], [5,6,h(nil),RW]]]]]
```

5.2.2 Reformatting and relinking

In the module `linkage2links.pl`, the linkage is reformatted: links and tokens are stored in dynamically asserted Prolog facts as illustrated in (44).

(44)
```
linkage2links:token(/////, 0).
linkage2links:token('cp[?]', 1).
linkage2links:token('copies', 2).
linkage2links:token('the', 3).
linkage2links:token('files', 4).
linkage2links:token('.', 5).
linkage2links:token(/////, 6).
linkage2links:token([], 7).
linkage2links:token([], 8).

linkage2links:link(7, 2, 'INFL', [s]).
linkage2links:link(3, 4, 'D', [m,c]).
linkage2links:link(2, 3, 'O', [p]).
linkage2links:link(7, 8, 'S', [s]).
linkage2links:link(8, 1, 'D', [s]).
```

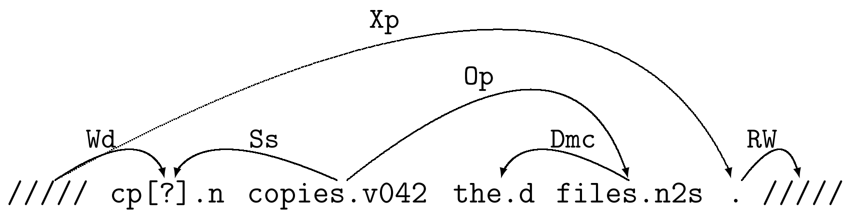
For each token an AVM is instantiated by `linkage2links.pl` in the module `features.pl`. This AVM so far stores the information on the position of the token in the sentence (beginning with position number 0) as well as the lexical head (word form) and the information received from the tag of the token (part of speech). (45) gives the AVM for the token *files* from the example.

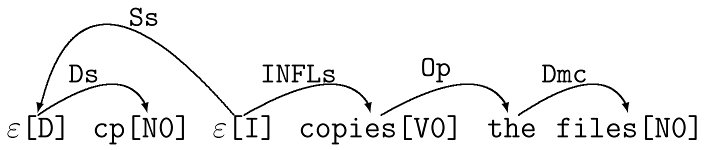
(45)
```
features:feature(1, position=4).
features:feature(1, hd=files).
features:feature(1, cat=n).
features:feature(1, bar=0).
```

(44) above shows that some links have been corrected during the process. The module `linkage2links.pl` has done the so-called relinking along the rules defined in `linkfeatures.pl` and has altered the original link structure (46) to a corresponding link structure (47) which is easier to convert into the desired constituency structure.

Relinking rules can be defined in the rule set of the program; they are not inherent to the converter. Relinking rules are correction rules for certain link types (or certain link types in certain environments, respectively). The user may want to define relinking rules for several reasons: (1) The user may want to correct strange links. Some links or link types returned by Link Grammar are rather peculiar, i.e. they differ from the common view of the corresponding syntactic construction. (2) The user may want to remove irrelevant links (e.g. those to the walls). (3) The user may want to alter links so that they are equivalent to a specific desired tree output.

In the example, the links are changed so that the conversion returns a tree where the functional category I (inflection) is present (i.e. a GB-like representation) and where DP-hypothesis is applied.

(46)


(47)


Note that links without direction (*Xp, RW* in our example) receive an arbitrary direction in the `linkage2links.pl` process, since `link/4` can only store directed links. If necessary, this kind of links must be corrected by an appropriate relinking rule stored in the rule set.[13] Irrelevant links and tokens have been omitted during the relinking process in `linkage2links.pl`.

5.2.3 Conversion

After `linkage2links.pl` the module `merge2tree.pl` is called. This module does the actual conversion of the link structure into a constituent structure. It therefore processes the information stored in the link types and their subscripts. The output of `merge2tree.pl` is a non-arranged constituent tree (48) as well as an updated set of feature structures belonging to the node identifiers of the tree structure. 'Non-arranged' means that only dominance is relevant in the structure but linear precedence (word order) is not yet considered.

(48)
```
[9, [10, [11, [12, [13, [1,'cp[?]']]], [8,[]]]], [14, [15, [16, [17,
[18, [19, [20, [4,'files']]], [3,'the']]], [2,'copies']]],
[7,[]]]]
```

5.2.4 Word order

Finally, the module `nicetree.pl` rearranges the constituents of the tree so that they follow the word order of the original sentence and dominance relations are still kept.

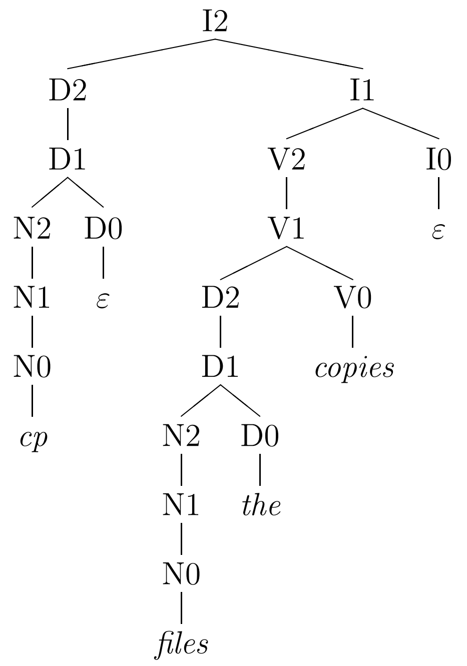
The rearranging of the constituents is illustrated below: (49) represents the non-arranged tree and is thus equivalent to tree structure (48) above, whereas (50) corresponds to the arranged tree structure in (51).

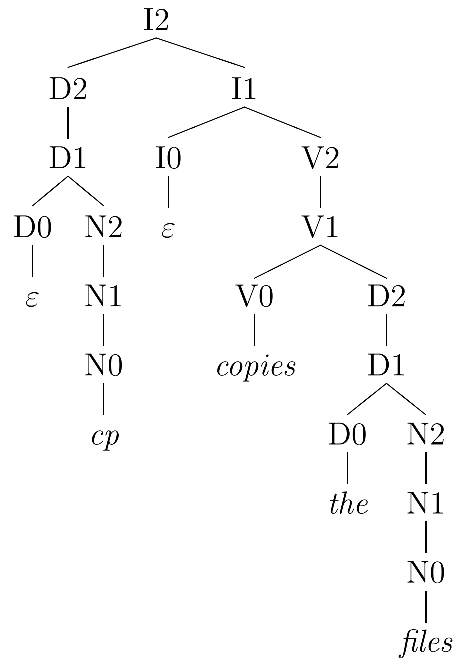
Note that for reasons of simplicity, only part of speech (category) and bar level are represented in the node labels of the two trees below.[14] The underlying node identifiers, however, may refer to much richer feature structures.

[13]For our example, `linkage2links.pl` returns the following warnings to the screen:

```
Link type Xp with directionality h(nil) arbitrarily changed into left-head-link!
Make sure that a correction-rule for this link type is provided in
linkfeatures.pl!
Link type RW with directionality h(nil) arbitrarily changed into left-head-link!
Make sure that a correction-rule for this link type is provided in
linkfeatures.pl!
```

[14]This is exactly what the predicate `nodelabels/2` in the module `nicetree.pl` does for reasons of graphical display: The node identifiers in the tree structure are replaced by node labels that express the category and the bar level of the node.

(49)


(50)


5.2.5 Post-processing

Finally, the format of the constituent structure is altered. `id/3` and `lp/3` facts are generated from the list notation (51) of the tree structure and `feature/3` facts are asserted, extending the `feature/2` facts by the number of the parsed sentence.

5.2.6 Output

The final output of the program therefore consists of a set of `id/3`, `lp/3` and `feature/3` facts as (partially) shown in (52). The list notation of the constituent tree (51), however, is also returned in the last argument of `linkage2constituents/4` or `link2tree/3` respectively.

(51)
```
[9, [10, [11, [8,[]], [12, [13, [1,‘cp[?]’]]]]], [14, [7,[]], [15,
[16, [2,‘copies’], [17, [18, [3,‘the’], [19, [20,
[4,‘files’]]]]]]]]]
```

(52)
```
link2tree:id(0, 9, 10).
link2tree:id(0, 10, 11).
link2tree:id(0, 11, 8).
link2tree:id(0, 11, 12).
link2tree:id(0, 12, 13).
link2tree:id(0, 13, 1).
link2tree:id(0, 9, 14).
link2tree:id(0, 14, 7).
link2tree:id(0, 14, 15).
link2tree:id(0, 15, 16).
link2tree:id(0, 16, 2).
link2tree:id(0, 16, 17).
link2tree:id(0, 17, 18).
link2tree:id(0, 18, 3).
link2tree:id(0, 18, 19).
link2tree:id(0, 19, 20).
link2tree:id(0, 20, 4).

link2tree:lp(0, 10, 14).
link2tree:lp(0, 8, 12).
link2tree:lp(0, 7, 15).
link2tree:lp(0, 2, 17).
link2tree:lp(0, 3, 19).

...
link2tree:feature(0, 9, position=1.5).
link2tree:feature(0, 9, hd=[]).
link2tree:feature(0, 9, cat=i).
link2tree:feature(0, 9, num=sg).
link2tree:feature(0, 9, bar=2).
link2tree:feature(0, 10, position=0.5).
link2tree:feature(0, 10, hd=[]).
link2tree:feature(0, 10, cat=d).
link2tree:feature(0, 10, num=sg).
link2tree:feature(0, 10, bar=2).
...
```

6 The module `link2tree.pl`

The module `link2tree.pl` fulfills three functions. First, as the main module of the program, it controls the conversion process, i.e. it takes the input handed over to it by the user or another program and returns the output after calling the appropriate modules. As such, it provides the two main interfaces of the program: `linkage2constituents/2` and `link2tree/3` (see 4.3). Second, it provides the user interface for the option handling (`chopt/1` and `echo_option/1`). This allows the user to alter the program options. Third, `link2tree.pl` is responsible for the displaying of the result with `draw.pl` (`display_tree/1`).

6.1 The main predicates

6.1.1 `linkage2constituents/4`

The predicate `linkage2constituents/4` is the predicate to be called by ExtrAns if the implementer wants to convert ExtrAns/Link Grammar linkages into constituent structures. As its first three arguments, it takes (i) the number of the parsed sentence, (ii) the set of linkages for that sentence in the form in which it is returned by ExtrAns, and (iii) the number of the linkage that shall be converted. In its fourth argument it returns the resulting constituent tree in the form of a Prolog list. Listing (53) shows the predicate and the way it calls the relevant predicates from other modules: First the predicate `linkage2links/2` from module `linkage2links.pl` is called to do the reformatting of the linkage. Second, `get_tree_of_links/1` (module `merge2tree.pl`) does the actual conversion into a constituent tree, and third, `wordorder/2` from module `nicetree.pl` is called to arrange the resulting tree according to the original word order of the sentence. Finally, the format of the resulting constituent tree is altered by `postprocess/2`.

(53)
```
linkage2constituents(SentNr, Linkages, ResultNr, ConstituentTree):-
    linkage2links(Linkages, ResultNr), !,
    get_tree_of_links(UnarrangedTree), !,
    wordorder(UnarrangedTree, ConstituentTree), !,
    postprocess(SentNr, ConstituentTree).
```

6.1.2 `link2tree/3`

An alternative method to use the converter is to start it up using the startup script as described in appendix A. This implies that the ExtrAns main module `main.pl` is consulted in `link2tree.pl` and the predicate `start_parser/0` is called at its beginning (54).

(54)
```
:-consult([main]).
:-start_parser.
```

If the converter is used in the described way, `link2tree.pl` is not called from ExtrAns but ExtrAns is called from `link2tree.pl`. This may be useful for testing purposes as long as one does not want the converter to be an integral component of ExtrAns.

The predicate `link2tree/3` allows the user to submit queries on the screen in the way illustrated in (55).

(55)
```
?- link2tree(‘‘cp copies the files.’’, 0, Result).
```

Listing (56) represents the predicate `link2tree/3`. The predicate first makes ExtrAns parse the sentence in its first argument by calling `parse/2` from `main.pl`. `print_links/2` displays the returned Link Grammar linkage(s) on the screen. Afterwards, `linkage2constituents/4` is called to do the conversion as described above. The number in the second argument of `link2tree/3` specifies the linkage to be converted. The resulting constituent tree is displayed on the screen by `display_tree/1`.

(56)
```
link2tree(String, ResultNr, ConstituentTree):-
        parse(String, Linkages), !,
        print_links(Linkages), !,
        linkage2constituents(0, Linkages, ResultNr, ConstituentTree), !,
        display_tree(ConstituentTree),
        !.
```

6.2 Option handling

The module `link2tree.pl` provides two predicates that allow the user (or parent processes) to check and/or alter certain program options. These options and their values are stored in the database file `linkfeatures.pl`. Two kinds of options are to be distinguished: Options that take Boolean values (either 0 or 1) and options that take non-Boolean values. In `linkfeatures.pl` a standard value is defined for every program option.

6.2.1 `echo_option/1`

The predicate `echo_option/1` can be used to check the current value of an option. It calls the predicate `if_option/1` from `linkinfo.pl`, the interface to the database. Options are stored in the form `option(Option=Value)`, which means that `echo_option/1` has also to be used like `echo_option(Option=Value)`. It is described in section 13 what particular program options are provided by Link2Tree at its current state.

6.2.2 `chopt/1`

The predicate `chopt/1` is used to alter the value of an option on the screen. It can be called as `chopt(Option=NewValue)` or as `chopt(Option)`. The former can be used for all options, the latter form is restricted to options that take Boolean values. It switches their value from 0 to 1 or from 1 to 0, respectively. In any way `chopt/1` calls the predicate `change_option/1` from `linkinfo.pl`.

6.3 Graphical display

6.3.1 `display_tree/1`

The predicate `display_tree/1` is responsible for graphically displaying the resulting constituent tree on the screen. It is only called if `link2tree/3` is called by the user. Then, it displays the result as described in 4.2.

By calling `nodelabels/2`, the predicate replaces node identifiers by category labels as already mentioned in 5.1.5. Finally, `use_draw/1` is called to apply Mark Holcomb's module `draw.pl` to the tree and induces its graphical display.

6.3.2 `use_draw/1`

The function of the predicate `use_draw/1` is to reformat the labeled constituent tree so that `draw.pl` can deal with it. This means that it has to be converted from a list form like `[np, [det, the], [n, dog]]` to a form where the node is the functor, like `np(det(the), n(dog))`. To achieve this, the predicate makes use of the built-in predicate "univ", which is realized as `=../2` in Prolog. The auxiliary predicates `recursive_univ/2` and `list_univ/2` redefine it recursive and for list handling. After the conversion `use_draw/1` calls `draw/1` from `draw.pl`.

6.4 Post-Processing

6.4.1 `postprocess/2`

The predicate `postprocess/2` alters the internal representation of the resulting constituent structure (consisting of a Prolog list denoting dominance and precedence relations and a set of `feature/2` facts for the feature structures of the nodes) into its final output format (consisting of sets of `id/3`, `lp/3` and `feature/3` facts, which are asserted dynamically in `link2tree.pl`).

At its beginning, `postprocess/2` retracts all previous `id/3`, `lp/3` and `feature/3` facts. This means, that the output facts of a conversion have to be imported in the parent module before the next conversion is started.

The actual conversion of the format is done by the predicates `tree2id/2`, `tree2lp/2`, `add_sentnr_to_features/1`. After these predicates have been called, the newly asserted facts are listed on the screen by calling the built-in predicate `listing/1` for them.

6.4.2 `tree2id/2`, `tree2lp/2`, `add_sentnr_to_features/1`

The predicates `tree2id/2` and `tree2lp/2` go recursively through a constituent tree in the form of a Prolog list and generate dynamically asserted `id/3` and `lp/3` facts to describe all the relations of immediate dominance (ID) and linear precedence (LP) in that tree. Note that `lp/3` only denotes *immediate* linear precedence. Three sister nodes `A, B, C` are thus grouped as `lp(SentNr, A, B)` and `lp(SentNr, B, C)`.

The predicates `tree2id/2` and `tree2lp/2` take the number of the parsed sentence as their first and the constituent tree in the form of a Prolog list as their second argument. The sentence number is delivered to the first argument of the `id/3` and `lp/3` facts.

The predicate `add_sentnr_to_features/1`, the only argument of which is the number of the parsed sentence, generates a `feature/3` fact for each of the `feature/2` facts stored in the module `features.pl`. The first argument of the newly generated fact contains the sentence number, the second argument the node identifier and the third argument the feature-value pair.

7 The module `linkage2links.pl`

The two main tasks of the module `linkage2links.pl` are the reformatting of the input linkage so that it can easily be handled by the converter and the correction of the linkage in order to make it more suitable for the transformation into constituent structures. As the main predicate of the module, `linkage2links/1` starts the processes to fulfill these tasks. The first three predicates it calls are concerned with preparatory tasks, the next three predicates reformat the input linkage and the last predicate starts the relinking process.

7.1 Preparation of the reformatting

7.1.1 `retract_all_dynamics/0`

There are three modules in Link2Tree that assert Prolog terms dynamically. These are the modules `linkage2links.pl, features.pl, linkfeatures.pl`. The module `linkfeatures.pl` stores program options. These may not be retracted at the beginning of each query but rather stay the same during the whole session, if they are not deliberately altered by the user (or the parent process respectively). The Prolog terms stored by `linkage2links.pl` and `features.pl`, however, belong to the current query only and must be removed before (or at) the beginning of the next query. As the feature handling module `features.pl` stands outside the linear course of the conversion process but rather provides tools to be called whenever needed, the point where the retraction must take place can not be defined there. It could be argued that the main module `link2tree.pl` was the right place to retract all dynamically asserted terms, namely at the beginning of `linkage2constituents/4`. I hold against it that all dynamic terms would have to be imported into `link2tree.pl`, NB a module which is close to the program interface. Therefore, for reasons of programming security, I have decided to do the retraction of dynamically asserted terms at the beginning of `linkage2links/1` by calling `retract_all_dynamics/0`. This entails that the predicates defined as dynamic in `features.pl` are imported into `linkages2links.pl`.

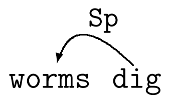
7.1.2 `get_xth_result/3`

The Link Grammar output for a sentence can consist of not only one but a whole set of possible linkages. This set of linkages is returned as a Prolog list, each element of which being itself a Prolog list representing a possible linkage for the sentence. Given several linkages LG provides certain heuristics for choosing the best parse i.e. the one to output first. A sophisticated cost system e.g. prefers the linkage in which the total length of links is lowest. Unfortunately, the survey over their cost system Temperley et al. (2000) provide in their documentation to Link Grammar is not detailed enough to get any precise insight of the heuristics applied. It is relevant for our purposes, however, that the LG/ExtrAns parser/post-processor output the most likely linkage first.

The predicate `get_xth_result/3` is called at the beginning of `linkage2links/1`. It extracts the linkage specified by its number (beginning with 0) in the first argument of the predicate. If there is a negative number in that argument an error message is printed to the screen and the conversion process is halted.

7.1.3 `get_lists/3`

The predicate `get_lists/3` prepares the reformatting of the linkage by splitting the input linkage into a list of tokens and a list of links. Tokens are stored as single elements in the LG token list, the order of which represents the word order in the sentence. Links are stored as lists of four elements. The first two of them denote the number of the linked tokens. The third element indicates if the link is directed to the left (`h(r)`), to the right (`h(l)`) or if there is no direction specified (`h(nil)`). The last element represents the link type and its subscripts. The linkage for example (57) is thus split into token list (58) and link list (59).

(57) 

(58) `['worms','dig']`

(59) `[[0,1,h(r),'Sp']]`

7.2 Reformatting

The purpose of reformatting is to have an easier and more flexible access to the data stored in the token and link lists. Both, tokens and links are to be represented as individual Prolog facts rather than to be packed in a long Prolog list. Links are intended to be expressed as `link(HeadID, DependentID, LinkType, LinkTypeSubscripts)`. Tokens are to be stored as `token(TokenID, Token)`.

7.2.1 `transform_link/2`

The predicate `transform_link/2` is called via `create_list_of_links/2` from the predicate `linkage2links/1`. It changes the format of the link items in the link list from lists to `link/4` terms as shown in (60).

(60) `[0,1,h(r),'Sp']` ⇒ `link(1,0,‘S’,[p])`

The predicate distinguishes three cases concerning the directionality of the link to be transformed: (i) the head is left `h(l)`, (ii) the head is right `h(r)` and (iii) the link has no direction at all `h(nil)`. In the first two cases either the first or the second token identifier in the list has to become the first argument of the `link/4` term. In the third case, if there is no directionality, one token is arbitrarily taken as the head of the link (since `link/4` can only store directed links because the first of its argument is defined as the head of the link and the second as its dependent). If this case happens a warning is printed on the screen and the user is asked to make sure that a corresponding correction-rule to be applied during the relinking process is provided in `linkfeatures.pl`.

7.2.2 `linkinfo:split_linktype/3`

It is the function of the predicate `split_linktype/3` to divide the information stored in the label of the link into two separate terms, namely a term for the link type (in upper-case letters) and a list of lower-case subscripts. This separation is crucial for the further processing of the information stored within link types and their subscripts. If both were melted to one single term a blow-up of the rule set would be the consequence since a separate rule would be required for every possible combination of link type and subscript(s).

The predicate `split_linktype/3` is imported into `linkage2links.pl` from the module `linkinfo.pl`, the interface to the rule set.

7.2.3 `assert_links/1`, `assert_tokens/1`

Whereas links are dynamically asserted by `assert_links/1` from the new link list in the form they have assumed in `transform_link/2`, `assert_tokens/1` calls the separate predicate `add_token/3`, which takes a token and a list of feature-value pairs as its input and returns the identifier of the AVM it has initialized for that token. Since Link Grammar assigns identifiers to the tokens by numbering them in the order they occur in the sentence, token identifiers correspond to their positions in the sentence at this stage of the process.[15] Therefore, `assert_tokens/1` puts the position of each token into the feature list handed over to `add_token/3` for that token. Apart from the feature POSITION, the feature CAT is initialized at this point. As the value of CAT must later be extracted from the tag of the tokens or the link types it is connected with, CAT is initialized with the default value ? at this point. The initializing of CAT for every token is crucial for the proper working of the converter.

`add_token/3` creates a new AVM for the token and adds the information it receives from the tag of the token by calling `get_tagfeatures/2` from `linkinfo.pl`. It furthermore adds the feature HD, which contains the lexical head of the node. The lexical head, i.e. the word form of a token without its tag, is returned in the third argument of `get_tagfeatures/3`. The label HD for the feature, containing the lexical head of a node, is program inherent. This feature must be defined as a head feature (cf. 12.3.4) in order to be transported to any mother node of which the token is the lexical head.

Last, the predicate `add_token/3` asserts a Prolog fact denoting the token in the form `token(Wordform, TokenID)`.

7.3 Relinking

The relinking process is started after the linkage has been reformatted by calling `do_relinking/0`. It provides a kind of preprocessor to the converter. Relinking corrects the input linkage according to rules declared in the rule set module `linkfeatures.pl`. `linkage2links.pl` controls the order in which these rules are called. It furthermore provides the syntax for the relinking rules in `linkfeatures.pl`. Relinking rules are Prolog rules the head of which is `relink:-`. Readers are referred to part III to learn more about the syntax of relinking rules.

7.3.1 `do_relinking/0`

The predicate `do_relinking/0` is a failure-driven loop that calls one `relink` rule after the other and terminates if there is no rule left to be applied (61). The below example shows that the same rule is called more than once if there are still links left to which it can be applied.

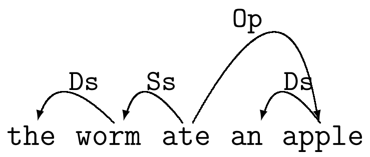
[15]It will change with the relinking process.

(61)
```
do_relinking:-
    call_rule,
    fail.
do_relinking.
```

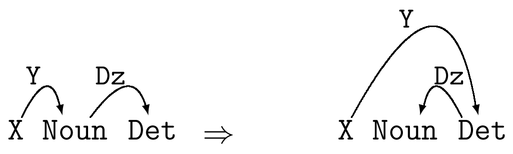
7.3.2 An example of relinking

As an example to illustrate what happens when `do_relinking/0` is called, let us assume that one wants to integrate DP-hypothesis into the conversion from linkage to constituent tree.[16] A simple sentence like (62) as parsed by Link Grammar is given in (63).

(62) *The worm ate an apple.*

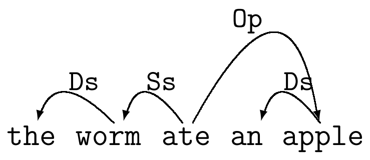
(63) 

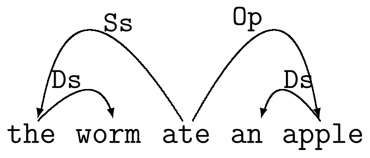
If one wants to follow DP-hypothesis, the fact that the nouns *worm* and *apple* should be dependents of their determiners has to be taken into account. (64) provides a general rule to correct such linkages in the explained sense of DP-hypothesis. (The letters X, Y and z denote variables.)

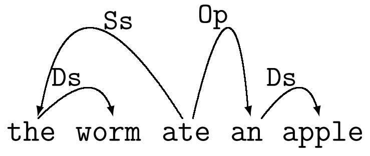
(64) 

Rule (64) says that if there is a *D* link from a noun to a determiner, the direction of this link must be changed, and if there is a link, the dependent of which is the said noun, it must be changed so that the determiner linked to that noun is its new dependent. If rule (64) was called by `do_relinking/0` it would first be applied to *the worm.* It would correct the linkage from (65) to (66) and then fail. By backtracking the same rule would be called again and be applied to *an apple.* It would alter the linkage from (66) to (67) and then fail. When the same rule would then be called again by backtracking it would fail since it could not be applied to any elements of the sentence anymore. At this stage `do_relinking/0` would terminate (or move on to the next rule if there were any of them left).

[16]For explanatory reasons I have to anticipate at this point as it has not yet been stated how a constituency based hypothesis is interrelated with a link structure. This will only become evident in the next section.

(65) 

(66) 

(67) 

The observation of the above example leads to the conclusion that the order in which the individual correction rules are called can be relevant. This is discussed in part III. As shown in section 5.2, the relinking rules may also insert new tokens. In the example given in section 5.2 an empty determiner and an empty inflection are inserted and linked.

7.4 Re-Arranging Links

If a link is added during the relinking process, it is asserted at the end of the `link/4` facts: the predicate `add_link/4`, which is responsible for the adding of links in relinking, makes use of `assertz/1`. However, the order in which the links are listed is not irrelevant. Link Grammar linkages list those links first, the left end of which is further to the left. If two links have the same left end, Link Grammar takes first the link with the right end further to the right. After having read the next section, one will understand that this is exactly the order the Link2Tree converter needs, since it implements a top-down algorithm (see section 8).

Relinking, however, may interfere with this order since it adds all new links at the end of the list. This may cause wrong constituent structures, namely structures that are not projective, i.e. trees with crossing branches (see 14.2.5). It is therefore necessary, that the aforementioned order is reestablished after the relinking process. This task is performed by the predicate `do_linksorting/0`, which is called right after `do_relinking/0`.

The predicate `do_relinking/0` first calls `links2lists/1`. This predicate reads all `link/4` facts into a list, each element of which is a link, which again is represented by a list of the form `[HeadID, DependentID, Type, Subscripts]`. The `link/4` facts are retracted as soon as the link they represent is read into the list. This list of links is then sorted by an adaptation of the quicksort algorithm: the predicate `linksort/2` rearranges the elements of the list in the above-mentioned order. Within this, the predicate `linksplit/2` is the extended pendant to the common `split/2` of the quicksort algorithm.

Last, the sorted list of links returned by `linksort/2` is turned back into `link/4` facts by the predicate `list2links/1`.

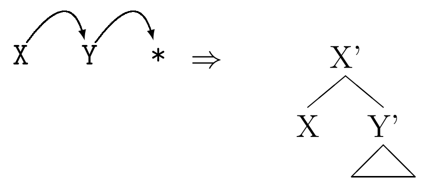
8 The module `merge2tree.pl`

8.1 Types of projection

8.1.1 A basic conversion operation

In order to come to a strategy for the conversion of linkages to constituent trees, in a first step, one may check if the notion of constituents is recognized in dependency grammar at all. Covington (1994:2) states that in DG constituents are a defined rather than a basic concept: "The usual definition is that a constituent consists of any word plus all its dependents, their dependents, and so on recursively." Tesnière (1959) calls such a constituent a 'noed' or 'nucleus' respectively. This works like in constituency: Nuclei have heads and can consist of nuclei themselves. From this concept one can imply a first most basic conversion operation (68).

(68)

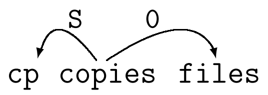


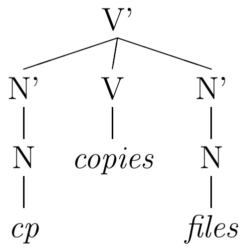
Note that in the above example, the convention of using a triangle to represent a constituent (subtree), the internal structure of which one is not concerned with, has been adopted. In the same way, an arc directed to the wild card $*$ has been chosen to denote a sub-linkage, or – in the terms of the above definition – a word plus all its dependents, their dependents, and so on recursively.

The basic operation in (68) implies that linkages are trees themselves (dependency trees). The condition that linkages must be trees places several constraints on the structure of the linkages: (i) there must be a root word in the linkage, from which every token of the linkage can be reached, (ii) the linkage has to be directed, (iii) acyclic, (iv) projective, and (v) every dependent can only have one head. These constraints for Link Grammar linkages have been discussed in section 2.

On the basis of (68) Covington (1994:3) interprets dependency grammar as equivalent to a particular strict form of X-bar theory in which (i) there is only one non-terminal bar level (i.e. X and X', but not X"); (ii) apart from bar level, X and X' (immediately dominating X) cannot differ in any way, because they are "really" the same node; (iii) there is no stacking of X' nodes (an X' node cannot dominate another X' with the same head).

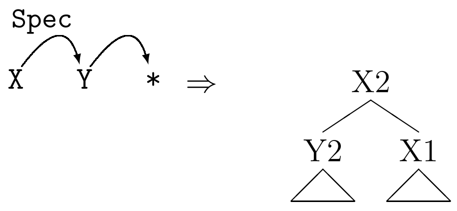
It will be demonstrated later how Link2Tree can be made to return constituent trees that comply with these restrictions. Operation (68) converts the linkage of sentence (69) to the constituent tree in (70).[17]

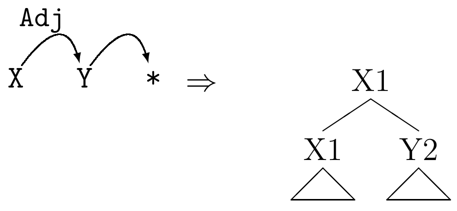
(69) 

(70) 

8.1.2 Conversion operations extended and generalized

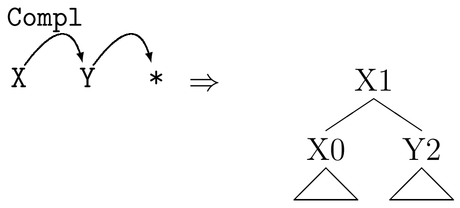
Covington (1994) shows the difficulties of the above interpretation of dependency grammar as a particular strict form of X-bar theory. Dependency trees cannot preserve the distinction between X, X' and X" which GB theory and GPSG use to distinguish between complements, adjuncts, and specifiers. Covington (1992:3) proposes "that complement, adjunct, and specifier be treated as three kinds of dependency, and labeled as such in the D-tree [dependency tree]." By following Covington's proposal, one can extend the conversion operation to (71–73).[18]

(71) 

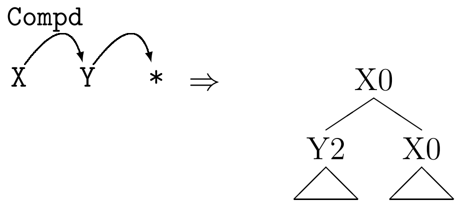
(72) 

[17]Note that the subject of the sentence is in its VP-internal position in (70), which corresponds to the recent assumption made by the so-called 'VP-internal subject hypothesis'.

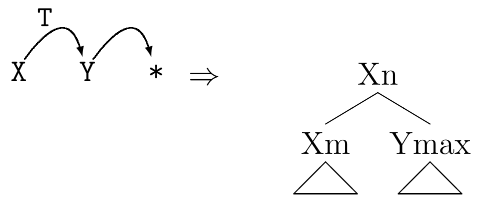
[18]Note that for reasons of accuracy the bar level is indicated by numbers rather than bars now. As for the order of constituents in the above examples, it is – although irrelevant for our current purpose – adapted to English word order.

(73) 

In addition to rules (71–73) one may even want to define a rule for the treating of compound nouns, like (74).

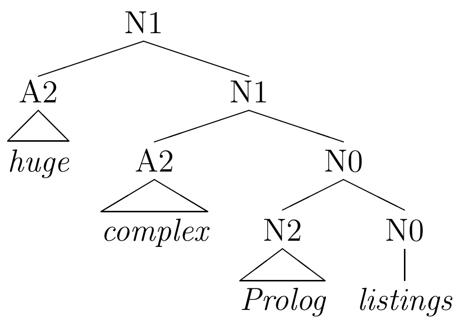
(74) 

To summarize the above conversion operations one can say that the operations defined for the converter must follow a most generalized form, given in (75).

(75)  where T=⟨m, n⟩

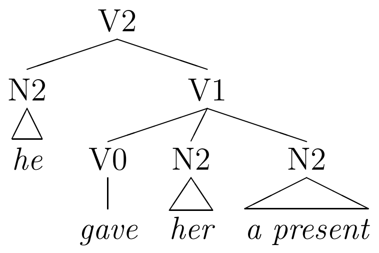
8.1.3 Recursiveness: stacked vs. flat structures

Covington (1994) states that one particular weakness of the most basic interpretation of dependency grammar given in 8.1.1 is its restriction to non-stacked structures, i.e. a node cannot dominate another node with the same head and bar level. One may, however, want to allow stacked structures, e.g. for multiple adjuncts or for compounds, like in (76).

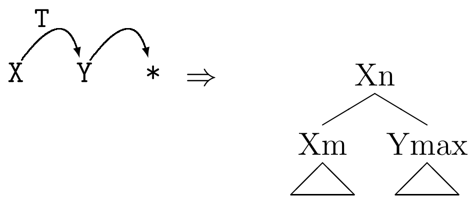
(76) 

On the other hand, one may also want to provide the facilities for a flat analysis, e.g. for double objects like in (77).[19]

[19]The scope of this section is the discussion of functionality. Readers interested in the pros and cons of particular analyses are referred to part III. For a introductory discussion of the sense of intermediate categories and stacked structures readers a referred to Covington (1994:4ff.) and Schneider (1998a:21ff.).

(77)


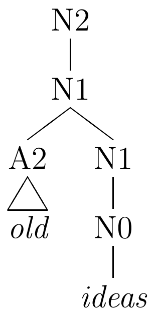
To do the conversion, it must thus be known if the type of projection produces stacked or flat structures or if it is not recursive at all. This information must be contained in the dependency type. Therefore, (78) is proposed. r stands for the aforesaid information on the recursiveness of the projection type. It takes either the value STACK if the type produces stacked structures, or FLAT if it allows flat structures, or NIL if it is not recursive at all, i.e. if it produces binary structures and cannot be stacked.

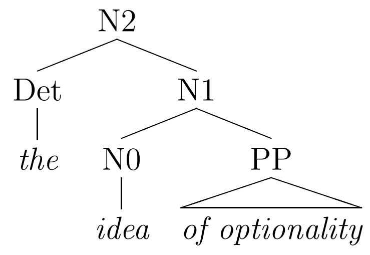
(78)


where
T = ⟨r, m, n⟩
r ∈ {STACK, FLAT, NIL}

8.1.4 Obligatory vs. optional projection types

One can distinguish obligatory from optional projection types. In common GB theory specifiers and complements are obligatory whereas adjuncts are optional. This means that an X1 node is inserted even if there is no complement present or that an X1 must be projected to an X2 node even if there is no overt specifier, like in (79). As shown in 3.6, the slots opened by empty specifiers serve as landing sites for certain moved constituents. On the other hand, no extra X1 node has to be inserted if there is no adjunct, like in (80).

(79)


(80)


In (81) the set represented by T in (78) above is extrended by an element o, which indicates optionality.

(81) T = ⟨r, o, m, n⟩
where
r ∈ {STACK, FLAT, NIL}
o ∈ {OBLIGATORY, OPTIONAL}

8.1.5 Ranking of the projection types

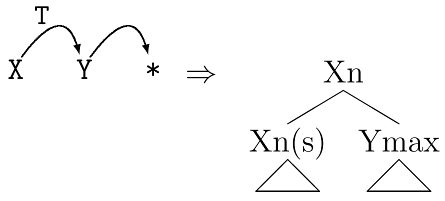
Covington (1994:7) gives a sketch of a reinterpretation of dependency grammar, which is consistent with current X-bar theory. It can be recognized as a prototype of a conversion algorithm:

> Given a head (X) and its dependents, attach the dependents to the head by forming stacked $\overline{X}$ nodes as follows:
>
> 1. Attach subcategorized complements first, all under the same $\overline{X}$ node. If there are none, create the $\overline{X}$ anyway.
> 2. Then attach modifiers, one at a time, by working outward from the one nearest the head noun, and adding a stacked $\overline{X}$ node for each.
> 3. Finally, create an $\overline{\overline{X}}$ node at the top of the stack, and attach the specifier (determiner), if any.
>
> (Covington, 1994:7)

From this description one can gain insight into the behavior of the algorithm for flat/obligatory projection types (complements), stacked/optional ones (modifiers) and for non-recursive/obligatory ones (specifiers). Furthermore, it is evident that in an algorithm another feature is relevant for projection types: their ranking. The above algorithm knows that it has to process the links in the order of the ranking of their projection types: complements first, then modifiers, specifier last.

So as to allow the user to define the form of X-bar theory applied to the tree output of Link2Tree, it is not useful for the ranking of projection types to be inherent to the conversion algorithm. It should rather be added to the definition of projection types. Two new elements for T are defined: t, which represents the identifier of the type, and s, which denotes the next lower projection type, i.e. the next projection type to process by a top-down algorithm. The element m for the bar level of the head-subtree is no longer needed since it is already defined indirectly by s. The revised conversion operation is given in (82).

(82)



where
T=⟨t, r, o, s, n⟩
r ∈ {STACK, FLAT, NIL}
o ∈ {OBLIGATORY, OPTIONAL}

8.1.6 `linkfeatures:treelevel/5` and `toplevel/1`

In Link2Tree the form of X-bar theory that is applied to the conversion can be defined by the user in the rule set. The predicate `treelevel/5` corresponds to set T from (82). Its first argument is the identifier (name) of the projection type. The second argument denotes its recursiveness (stack|flat|nil) and the third its optionality (obligatory|optional). The fourth argument is the identifier of the next lower projection type. The last argument is a list of features, which are added to the node of the projection (e.g. the bar level), i.e. features that do not necessarily comply with the corresponding features of the head.

The restricted form of X-bar theory described in 8.1.1 would only need definition (83).

(83)
```
treelevel(projection, flat, obligatory, [], [bar=1]).
```

The more common form of X-bar theory which is represented in rules (71–73) above or in Covington's algorithm, quoted in the previous subsection, requires the three Prolog terms given in (84). To add the conversion rule for compound constructions, (74) above, the definition would have to be revised to (85).

(84)
```
treelevel(specifier, nil, obligatory, adjunct, [bar=2]).
treelevel(adjunct, stack, optional, complement, [bar=1]).
treelevel(complement, flat, obligatory, [], [bar=1]).
```

(85)
```
treelevel(specifier, nil, obligatory, adjunct, [bar=2]).
treelevel(adjunct, stack, optional, complement, [bar=1]).
treelevel(complement, flat, obligatory, compound, [bar=1]).
treelevel(compound, stack, optional, [], [bar=0]).
```

It will be shown in section 8.2.1 below that the conversion algorithm applied in Link2Tree is a top-down algorithm. To facilitate the conversion process, the projection type which is topmost in the ranking, is defined in the separate predicate `toplevel/1` in the rule set. For example (85) above, `specifier` must be defined as the topmost projection type, like in (86).

(86)
```
toplevel(specifier).
```

8.1.7 Projection types vs. link types

Note that in Link2Tree, projection types are not equivalent to link types. Every link type has a projection type, but many link types can have the same projection type. As already shown, projection types define the way a link (or a sub-linkage respectively) is converted to a subtree. Link types, on the other hand, contain much more information than just their projection type. They also provide information to be assigned to the AVMs of their head and dependent. Link types are stored in the rule set by the predicates `typefeatures/4`, the last argument of which denotes its projection type, and `subsfeatures/4`. They are accessed via the predicate `linkfeatures/5` in the module `linkinfo.pl`.

8.2 The conversion

8.2.1 The algorithm

The extensions and generalizations to Covington's conversion algorithm made in the previous section let us formulate a conversion algorithm for Link2Tree. The algorithm describes the way a linkage (or a sub-linkage respectively) is converted to a constituent tree, beginning with its root word.

> Given a head (X) and its dependents, attach the dependents to the head as follows:
>
> 1. Begin the conversion with the topmost projection type.

2. Look for a link of the aforesaid projection type, the head of which is the aforesaid token. If there is more than one such link take the one the dependent of which is furthest from the head.

   If the projection type is not overt, i.e. if there are no links of this projection type from a token in the linkage, do either

   (a) if the projection type is obligatory, project the token according to the projection type and then convert the remaining sub-linkage, i.e. the sub-linkage the root of which is the aforesaid token, to a subtree, beginning with the next lower projection type (2ff.), and attach the subtree to the projection of the token, or,

   (b) move on to the next lower projection type, if the projection type is optional.

3. Project the head according to the definition of the projection type.

4. Convert the dependent sub-linkage, i.e. the sub-linkage the root word of which is the dependent of the aforesaid link, to a subtree (1ff.) and attach this to the projection of the head.

5. If the projection type is recursive, i.e. if there can be more than one link of this projection type from the same head, convert the remaining dependent sub-linkages to subtrees (1ff.) and do either

   (a) attach them directly to the projection of the head, if the projection type is determined to produce flat structures, or,

   (b) if the projection type is determined to produce stacked structures, attach them to the projection of the head, one by one, by working inward from the one furthest from the head, and adding a stacked projection of the head for each.

6. Convert the head sub-linkage, i.e. the sub-linkage the root of which is the head of the aforesaid link, to a subtree, beginning with the next lower projection type (2ff.), and attach the subtree to the aforesaid projection of the head.

   If there is no lower projection type, attach the head as a leaf to its aforesaid projection.

8.2.2 The root link (`linkfeatures:rootlink/2`)

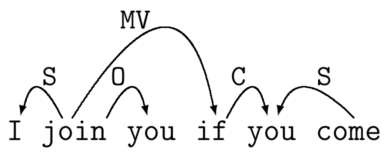
As stated in section 2.2.2.4, because of the lack of directionality in original Link Grammar linkages there is no self-evident root word. Nevertheless, it has become evident that one implication of the conversion operations developed above is that the linkage has the form of a tree, which again implies that there must be a root node, from which every leaf in the tree can be reached. In ExtrAns linkages, walls can serve as such root nodes. But since they are dummy elements (denoting the boundary of the sentence) rather than syntactically motivated, the concept of walls is abandoned for the conversion and the root word is chosen on a syntactic basis. Link2Tree defines the root word as the head of a so-called root link type. The user can define such root link types in the rule set. An *S* (subject) link might be the most common root link type to choose. If the user wanted to convert single noun phrases rather than whole sentences, other link types may be defined to denote the root word. The predicate `rootlink/2` in the module `linkfeatures.pl` takes a link type as its first and a list of required subscripts to the link type as its second argument. If one sticks to the subject link as root link, the appropriate Prolog fact might look like (87).

(87) `rootlink('S', _AnySubscripts).`

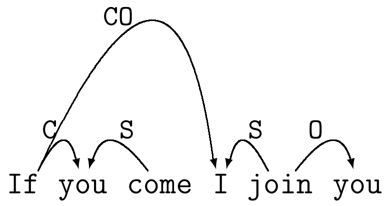
The possibility that multiple links of the root link type may be present in a more complex sentence must be considered. This potential problem can be solved in two ways: One possibility is to make sure that the link, the head of which is intended to be the root word, is the one processed first, i.e. the first link of this link type in the link list. Linkages list those links first, the left end of which is further left, and, if two links have the same left end, the one the right end of which is further right. If one supposes that *S* is to be the root link type, this method works for linkage (88), which is represented in a simplified way in (89), where subscripts and irrelevant links are not shown. Suppose, *S* was defined as root link type, the *S* link between *I* and *join* would be taken as the root link according to the aforementioned heuristics. As for the choice of the root link, linkage (89) does therefore not cause any problems.[20]

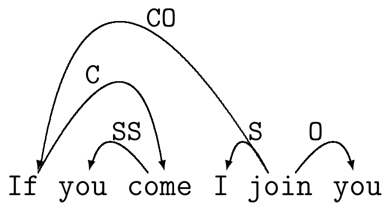
(88)
```
  +---------------------Xp---------------------+
  |           +------MVs----+                  |
  +-Wd-+-Sp*i-+---Ox--+     +-Cs-+---Sp--+     +-RW+
  |    |      |       |     |    |       |     |   |
///// I.p join.v061 you.p if.c you.p come.v011 . /////
```

[20]It is evident, that there is at least one other difficulty with linkage (89), however, since the token *come* cannot be reached from the root word *join*. One would therefore have to define a relinking rule which would make *come* the dependent of the *C* link.

(89) 

The method fails, however, for linkage (90). The *S* link of the subsidiary clause would be taken as the root link, instead of the one in the main clause. Such complications can be avoided by making sure that no other links of the same type occur in a linkage by defining appropriate relinking rules. The *S* link of a subsidiary clause may thus obtain another label in the relinking process to guarantee that the finite verb of the main clause is taken as the root word of the linkage, as shown in (91).[21]

(90) 

(91) 

The module `merge2tree.pl` is started up by the call of `get_tree_of_links/1`, which returns the constituent tree. This predicate extracts the root word of the linkage by calling `linkinfo:get_rootlink/2` and checking if there is a link of the root link type in the linkage. If this is the case, the predicate hands the head of this link over to `merge2tree/2`, which in turn returns the resulting tree structure to it.

8.2.3 Starting the conversion (`merge2tree/2`)

The predicate `merge2tree/2` initializes the conversion of a (sub-)linkage to a tree. If the category of the root word delivered to it is projectable, `merge2tree/2` determines the topmost projection by calling `get_toplevel/1` from the module `linkinfo.pl` according to step 1 of the algorithm in 8.2.1 and initializes step 2. As pointed out at the end of section 3.6, some constituency theories like GPSG claim the existence of non-projectable categories. In Link2Tree, the user can define which category is projectable by applying the predicate `projectable/1` in the rule set. In `merge2tree/2` the projectability of a token is

[21]Note that also the problematic *CO* and *C* links have been relinked from (90) to (91). This has nothing to do with the finding of the correct root link.

checked by calling `cat_is_projectable/1`. This predicate tries to locate the category of the token in its feature matrix, and if there is no feature CAT present in the AVM, tries to extract the category information from the topmost link the head of which is the aforesaid token. After having found the category of the token, it checks its projectability via the interface predicate `is_projectable/1` from module `linkinfo.pl`. The predicate fails, if no category can be found for the token. Non-projectable tokens are turned into a leaf, i.e. into a tree consisting of a preterminal and a terminal, by the predicate `add_leaf/2`, in the form of `[MinorCategory, Word]`.

In addition to this, if the category is projectable, `merge2tree/2` calls the predicate `trace/1`, which removes links, the head of which is the root word, and the projection type of which is `trace`. This is explained in more detail in 8.3. Trace links are a way to solve the problem of cyclic link types (like *B* links) and to represent movement.

8.2.4 The core of the conversion (`projection/3`)

The predicate `projection/3` is called at the end of the first `merge2tree/2` predicate. It distinguishes three cases, which comprise step 2 of the algorithm. The first case is called if a link of the demanded projection type and with the demanded head exists. The features from its link type and subscripts to be added to the head and the dependent of the link are extracted from the rule set via `linkfeatures/5` from module `linkinfo.pl`. The lists of features are added to the AVMs of the head and the dependent token by `add_features_to_both/4` and the link is removed. `projection/3` then triggers step 3 (`project(HeadNr, ProjType, ProjNr)`) and step 4 (`merge2tree(DepNr, DepTree)`) of the algorithm: a projection of the head is created and the dependent sub-linkage is converted to a subtree. Foot features are added to the AVM of the projection by `apply_ffp/2`. Finally, `recursive_projection/4` is called, which corresponds to step 5 of the algorithm.

The other two cases of `projection/3` are called if no link of the demanded projection type exists. They stand for steps 2a and 2b of the algorithm.

8.2.5 Projection in Link2Tree (`project/3`)

The predicate `project/3` creates a projection of a token according to the specified projection type. It first fetches the list of features defined in the projection type definition. It then projects the AVM of the token, creating a new AVM identifier and assigning copies of all head features and foot features of the token to it by calling `apply_hfp/2` and `apply_ffp/2` respectively. Features are defined as head features or foot features in the predicates `headfeature/1` and `footfeature/1` in the rule set. Last, the AVM of the projection is updated by the features defined for it in the projection type definition, e.g. the bar level is adapted in this way. Note that if there is a dependent subtree attached to the projection, the Foot Feature Principle (see also 3.5) is applied by calling `apply_ffp/2` in the parent predicate `projection/3`.

8.2.6 Recursive projection types (`recursive_projection/4`)

The predicate `recursive_projection/4` deals with recursive projection types, i.e. with projection types that can be realized more than once for the same head. It distinguishes three cases, the first two of which correspond to step 5 of the conversion algorithm. The reader is referred to section 8.2.1 for a description of their function. The third case of `recursive_projection/4` is the break condition of the predicate, which is called if no more links of the projection type exist or if the projection type is not defined as recursive. It calls `next_projection/3`.

8.2.7 Moving on in the ranking (`next_projection/3`)

At the point where all dependent subtrees of a head (the links of which are of the same projection type) have been treated, the conversion goes on with the link(s) of the next lower projection type. This happens in steps 2a, 2b and 6 of the algorithm. To move on to the link(s) of the next lower projection type, `next_projection/3` is called. It takes the previous projection type and the head as its first two arguments and returns the remaining subtree of the head in its third argument. `next_projection/3` makes use of the forth argument of `treelevel/5` to receive the information of the next lower projection type. It calls the interface predicate `get_treelevel/5` from `linkinfo.pl`.

If the bottommost projection type has already been reached, i.e. if the forth argument of `treelevel/5` is the empty list `[]`, the predicate attaches a leaf subtree (the word and its preterminal node) by calling `add_leaf/2`.

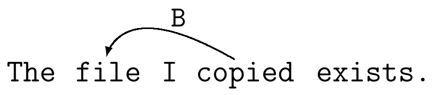
8.3 Trace links (`trace/1`)

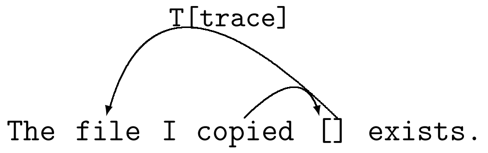
Section 8.2.3 explained that there is a program inherent projection type called `trace`. Links of this projection type are treated in a special manner: They are removed before the actual conversion of the sub-linkage they belong to. The head and the dependent of such a link obtain an additional feature `trace=TraceIndex` and remain interrelated in this way. Trace links can be used to solve the problem of the cyclic *B* links.

> *B* serves various functions involving relative clauses and questions. It connects transitive verbs back to their objects in relative clauses, questions, and indirect questions ("The DOG we CHASED", "WHO did you SEE?"); it also connects the main noun to the finite verb in subject-like relative clauses ("The DOG who CHASED me was black".)
>
> (Temperley et al., 2000)

Certain link types can thus be defined as trace links in order to exclude them from the conversion, since they violate the constraints of the dependency tree, but still preserve the information of the connectedness of their head and dependent.

The *B* link in (92), for instance, is an obstruction to the conversion of the sub-linkage according to the algorithm as it introduces circularity to it. However, one may want to keep the information on the relation the link expresses. The user may therefore define a relinking rule for *B* links in such environments that changes (92) into (93) during the relinking process.

(92)


(93)


The predicate `trace/1` is called to treat all the trace links of a head which is the argument of the predicate. It succeeds if no trace link exists with the given head. If there are trace links, they are removed, the features defined in the link type definition are added to the AVM of the head and the dependent. A new trace index is created and the special feature `trace=TraceIndex` is also added to both the AVM of the head and of the dependent. Note that the program inherent keyword `trace` must not be used to denote a user defined projection type in `treelevel/5`, neither can it be used for a user defined feature.

9 The module `nicetree.pl`

The constituent tree delivered by the conversion module `merge2tree.pl` represents dominance relations but does not consider linear precedence. Therefore, one function of the module `nicetree.pl` is to rearrange the constituents in the tree so that their order complies with the word order of the original sentence. The other function of the module is to provide a tool for the replacement of node identifiers by labels for category and bar level. Trees labeled in this way are used for the graphical displaying by `draw.pl`.

9.1 Linear precedence (`wordorder/2`)

The predicate `wordorder/2` takes the unsorted constituent tree delivered by the conversion module `merge2tree.pl` as its input in the first argument. It returns the sorted tree as the second argument. It sorts the constituents of the tree according to their feature POSITION. This feature is thus inherent to the program rather than user defined. As it will be shown below, in most cases it is generated automatically, and even where it is added by the user, namely if additional tokens are inserted by relinking rules, its value is calculated by Link2Tree.

9.1.1 Token positions

The feature POSITION is assigned to the AVM of a token by the predicate `assert_tokens/1` in `linkage2links.pl` (cf. 7.2.3). The position of tokens of the original input corresponds to the token identifier, which is generated by numbering the tokens in the order of their occurrence in the token list of the linkage, which is equal to their order in the sentence.

Since POSITION is defined as a head feature in the rule set, every projection of a token obtains the position of its head. Like this, the relative order of sister constituents can be guaranteed throughout the tree.

The position of tokens inserted by a relinking rule, however, must be carefully calculated to preserve the linear order of constituents. The module `linkage2links.pl` provides a set of predicates for the definition of the position of such a token in the correction rules in the rule set.

9.1.2 Recursive quicksort

The predicate `wordorder/2` first cuts the node (i.e. the first element) from the tree list it is given and sorts the remaining elements (the daughters of the node) according to the value of their feature POSITION.

For the sorting of the constituents the quicksort algorithm is applied. Its common Prolog realization has been adapted to the situation in Link2Tree: The comparison of two elements of the list to be sorted (here, the sisters of a local tree) is extended as shown in listing (94) since the position of the constituents rather than their node identifiers need to be compared.

(94)
```
split(_X, [],[],[]).
split(X, [Y|Tail], [Y|Small], Big) :-
    % common variant: X > Y, !, split (X, Tail, Small, Big).
    X=[XID|_], Y=[YID|_],
    feature(XID, position=PositionX),
    feature(YID, position=PositionY),
    PositionX > PositionY, !,
    split(X, Tail, Small, Big).
split(X, [Y|Tail], Small, [Y|Big]) :-
    split(X, Tail, Small, Big).
```

Since the daughter elements of a tree list can be tree lists themselves, the sorting has to be executed recursively. An auxiliary predicate `recursivesort/2` is implemented to introduce this recursion to the sorting process.

9.2 Labeled trees (`nodelabels/2`)

The predicate `nodelabels/2` recursively works through the tree structure given to it, similar to `wordorder/2` above. It replaces the identifier of the node of a tree by a label, and does so for the nodes of all its daughter trees, and so on recursively. `nodelabels/2` is only called from `link2tree/3` in `link2tree.pl` and serves the graphical display by `display_tree/1` only. It is not relevant for the further processing of the Link2Tree output. The predicates `matrixnr2label/2` and `recursivelabels/2` control the recursive execution of the labeling, similar to `quicksort/2` and `recursivesort/2` in the sorting above.

The predicate `add_catbar2label/2` extracts the value of the features CAT and BAR for every node and composes them to a node label. If the program option `bar` is set to `1`, the bar numbers are transferred into letters as defined in `bar/2` by `barnr2barletter/2`: X2 becomes XP, X1 becomes Xbar and X0 becomes X. If the program option `display_traces` is set to 1, the trace indices of the nodes are extracted and added to the label by `add_trace2label/3`. A labeled tree with visible traces output by `display_tree/1` may thus look like the extract shown in (95).

(95)

```
                      d2.t0
                        |
                        |
                      d1.t0
                        |
  +-----------------------+
d0.t0                     n2
  |                       |
  |                       |
  |                       n1
  |                       |
  |      +--------------------+
  |      n0                   c2
  |      |                    |
  |      |                    |
  |      |                    c1
  |      |                    |
  |      |      +----------------+
  |      |      c0               i2
  |      |      |                |
  |      |      |     +------------+
  |      |      |     d2           i1
  |      |      |     |            |
  |      |      |     |    +-----------+
  |      |      |     d1   i0          v2
  |      |      |     |    |           |
  |      |      |     |    |           |
  |      |      |     d0   |           v1
  |      |      |     |    |           |
  |      |      |     |    |       +-------+
  |      |      |     |    |       v0    d2.t0
  |      |      |     |    |       |       |
  |      |      |     |    |       |       |
  |      |      |     |    |       |     d1.t0
  |      |      |     |    |       |       |
  |      |      |     |    |       |       |
  |      |      |     |    |       |     d0.t0
  |      |      |     |    |       |       |
  |      |      |     |    |       |       |
 the    file    []    I    []    copied    []
```

If the program option `display_nodeid` is set to 1, the node identifiers are shown in front of the node labels in the tree. They are added to the label by the predicate `add_matrixnr2label/3`.

Note that if the user wants to make use of the graphical screen displaying supported by `nodelabels/2`, the features for category and bar level have to be called CAT and BAR in the rule set. Otherwise, they are not recognized by `add_catbar2label/2` and blank labels will be produced.

10 The module `features.pl`

The module `features.pl` is responsible for initializing, storing and updating the feature structures of the nodes in the constituent tree. It stores the features and their values as dynamically asserted Prolog facts `feature/2`, where the first argument is the node identifier and the second argument the feature-value pair. Feature-value pairs are represented as Prolog Terms of the form `Feature=Value`. All instantiations of `feature/2` that have the same node identifier in their first argument form together the AVM of that particular node.

10.1 Simple feature structures in Link2Tree

At its current state, Link2Tree only deals with simple feature structures. Simple feature structures take only non-complex values, i.e. values that are not feature structures themselves (cf. section 3.4). The values are intended to be simple Prolog terms – either atoms or numbers. Apart from storing the feature structures, the module `features.pl` provides a set of tools to initialize and update AVMs.

10.1.1 `create_matrix/2`

The predicate `create_matrix/2` initializes a new AVM. It first produces a new node identifier by calling `make_new_matrixnr/1`. This auxiliary predicate stores the last used node identifier in the predicate `last_matrixnr/1`. If it is called, it increases this number by 1 and returns it. The old number is replaced by the increased one in `last_matrixnr/1`. If there has been no node identifier used so far, the predicate initializes one with the value 0.

`create_matrix/2` returns the node identifier of the newly initialized AVM in its second argument. In its first argument it takes a list of feature-value pairs. These are added to the new AVM via calling `add_featurelist_to_matrix/2`.

10.1.2 `update_feature/2`

The predicate `update_feature/2` takes the node identifier of the AVM where a feature shall be updated as its first and the feature-value pair to be updated as its second argument. If the feature already exists in the AVM (which is usually the case, since the predicate is only called under such circumstances), `update_feature/2` replaces the existing value of the feature by the value in its second argument. If the feature does not exist yet, it is added to the AVM.

10.1.3 `add_featurelist_to_matrix/2`

The predicate `add_featurelist_to_matrix/2` adds the list of feature-value pairs specified in its second argument to the AVM specified by the node identifier in its first argument. It checks whether each feature of the list already exists in the AVM. If it does, the value of the feature is updated by calling `update_feature/2`. If the feature does not yet exist in the AVM, a new `feature/2` fact with the specified node identifier and feature-value pair is asserted dynamically.

10.2 Outlook: complex feature structures in Link2Tree

It may be desirable to provide facilities to handle complex feature structures in Link2Tree. Complex feature structures in constituent trees (cf. 3.4) are especially welcome in relation to unification based grammar theories, like HPSG, GPSG, and others. Gazdar and Mellish (1989:228) propose a broadly accepted Prolog notation for complex feature structures. For implementation purposes, they represent a feature structure as a list of feature-value pairs whose tail is left open as a variable. Müller (1998:143), who has adopted the method of Gazdar and Mellish (1989), points out that the open tail is important: it allows further feature-value pairs to be added without the necessity to create a completely new Prolog structure. In general, an AVM is represented as a Prolog list, the tail of which is a variable. Feature-value pairs are bundled into a single Prolog structure that can appear as an element of a list. Features and their values are connected by the operator ":", whose syntactic properties may be defined as `op(500, xfy, :)`.[22] Values that are simple are represented by a Prolog atom. Complex values are represented as a list of feature-value pairs with an open tail themselves. Feature structure (96) is therefore represented as (97).

(96) $$\begin{bmatrix} \textsc{pos} & \text{N} \\ \textsc{agr} & \begin{bmatrix} \textsc{num} & \text{Sg} \\ \textsc{pers} & 3 \end{bmatrix} \end{bmatrix}$$

(97) `[pos:n, agr:[num:sg, pers:3|_]|_]`

The integration of such facilities to Link2Tree would induce several revisions in the program code, not only in the module `features.pl` but also in the rule set, and wherever features

[22]Note that the predicate `:/2` sometimes conflicts with the module system in SICStus Prolog, especially if used together with Emacs. The use of `::/2` instead is therefore recommended.

serve as program inherent keywords (e.g. `cat` or `trace`). I suggest to do the following alterations if the user/implementer wants to expand Link2Tree in such a way.

As they are part of complex structures, features can no longer be represented by a simple label only. Paths may indicate the exact position of the feature in the AVM. The former feature-value pair `num=pl` may thus be represented as `[agr:[num:pl|_]|_]`. Feature lists given in the current rule set (cf. part III) would have to be replaced by complex feature structures. A feature list `[num=sg, pers=3]` given in a rule would thus be altered to the AVM `[agr:[cat:n, bar:0|_]|_]`. Therefore, the interface predicates provided in `features.pl` would have to be adopted to the new situation. Adding a feature list to an AVM would then mean adding an AVM to an AVM. This resembles the notion of *graph unification*, which is crucial to the treating of complex feature structures. For the purposes of Link2Tree, however, an expansion of this notion is necessary, as the unification must not happen in both directions but only in one, namely the existing AVM being updated by the AVM given in the rule. Furthermore, conflicting feature-value pairs must not make the unification fail but the existing value must rather be altered to the value delivered by the rule.

The core of an adopted module `features.pl` would therefore be the predicate `dag_update/3` given in listing (98), which takes an AVM in its first argument and updates it by the AVM given in its second argument, delivering the updated AVM in its third argument.[23] Different existing interfaces may be rewritten for it: `create_matrix/2`, `update_feature/2`, `add_featurelist_to_matrix/2`.

[23]The expression 'DAG' stands for 'directed acyclic graph', which is another description for complex feature structures.

(98)
```
% dag_update(+DagToUpdate, +UpdatingDag, -UpdatedDag)

dag_update(X, X, X):-!.
dag_update([F:V1|R1], [F:V2|R2], [F:V3|R3]):-
    !, % do not call next clause if pattern matched
    dag_update(V1, V2, V3),
    dag_update(R1, R2, R3).
dag_update([G:V1|R1], [F:V2|R2], M):-
    !, % do not call next clause if pattern matched
    dag_update(R1, [F:V2|_], R3),
    dag_update([G:V1|R3], R2, M).
dag_update(_V1, V2, V2).
```

The predicate `dag_update/3` updates complex feature structures as demonstrated in (99) and (100) below. In example (99), features that do not yet exist in the AVM are added to it. In (100) an AVM is updated by another AVM, which contains features that already exist in the original AVM. The example shows the projection mechanism, i.e. the changing of the bar level of a node.

(99)
```
| ?- dag_update([cat:n|_], [agr:[num:pl|_], bar:0|_], UpdatedAVM).

UpdatedAVM = [cat:n,agr:[num:pl|_A],bar:0|_B] ?
```

(100)
```
| ?- dag_update([cat:n,agr:[num:pl|_],bar:0|_], [bar:1|_], ProjectedAVM).

ProjectedAVM = [cat:n,agr:[num:pl|_A],bar:1|_B] ?
```

The predicate `feature/2` may be kept as an interface predicate to get the value of certain features. For the storage of the AVMs, however, I propose to define a predicate `avm/2`, the first argument of which is the node identifier and the second argument is the complete AVM of that node.

Wherever feature labels denote features in the existing Link2Tree, they would have to be replaced by complete path indications: `[agr:[num:NUM|_]|_]` instead of `num=NUM`.

Part III

The rule set

11 The rule set and its interface

The module `linkfeatures.pl` contains the rule set of Link2Tree. It comprises information on the way linkages must be converted to constituent structures, how they must be preprocessed (correction rules), and it stores the program options. Linguistically motivated adaptations, e.g. altering the structure of the output constituent trees, need only be made in this file.

The module `linkinfo.pl` provides interfaces to the rule set. The information stored in `linkfeatures.pl` can only be accessed through `linkinfo.pl`. The following sections describe what tools the interface module provides to handle particular information of the rule set.

Note that some declarations of the rule set must not be altered. They are relevant for a proper working of the converter. Labels for features and projection types may be freely chosen by the user – apart from a restrictred number of program inherent labels that must be used to guarantee a proper working of the converter. These obligatory labels comprise the feature labels CAT (8.2.3), POSITION (9.1.1), HD (7.2) as well as the projection type `trace` (8.3). The three features CAT, POSITION and HD are initialized automatically during the process. The use of the projection type `trace` depends on the syntactic model one chooses for the output constituent structure.

The module `linkfeatures.pl` imports special syntax predicates for relinking rules from the modules `linkage2links.pl` and `features.pl`.

12 Conversion rules

12.1 Tag features

In ExtrAns, tags are added to the tokens. The creators of Link Grammar only use tags to distinguish different lexicon entries for a word, such as `run.n` and `run.v` (the first stands for the noun *run*, the second for the verb). In Link Grammar, tags are thus only used if a particular lexicon entry is ambiguous. Schneider has extended this system of tagging and added word-class based tags to each word in the lexicon (except for multi-word entries). These tags contain information on the linguistic category and the subcategorizing of the words. Readers are referred to Schneider (1999) for a detailed description of this fine-grained tagging system. In Link2Tree, however, the user may not want to loose the information provided in the word tags. As shown in section 7.2.3, it is added as a set of feature-value pairs to the AVM of the respective token during the reformatting of the linkage. The user defines these tag features in the rule set predicate `tagfeatures/2`.

12.1.1 `tagfeatures/2`

Tags consist of a string of characters, each of which denotes a sub-category of the category that is specified by the character to its left. For the purposes of the Link2Tree rule set, however, they are represented as Prolog lists of atomic terms. The tag `.n2s`, for instance, denoting plural nouns ending in *s*, is represented as `[‘n’,‘2’,‘s’]`. One may want to assign to a word with the aforesaid tag the features CAT=N for the category, MASS=PL , in order to express that it is countable rather than a mass or a singular countable noun, and PLEND=S to denote its plural ending. One could do so by stating rule (101).

(101) `tagfeatures([‘n’,‘2’,‘s’|_], [cat=n, mass=pl, plend=s]).`

It is recognizable from (101) above that the first argument of `tagfeatures/2` denotes the list of tags and the second argument is the list of feature-value pairs to be added to the token. If the user is working with any form of X-bar theory the feature BAR=0 can be added to the list since tags always denote constituents on the lowest bar level. Note that there is an unspecified rest added to the list of tags. This enables Link2Tree to recognize subcategorizes of this particular tag class even if the user has not stated them explicitly. Declaration (102) is applied to words that take `.n` as a tag, but also to all sub-categories of this tag class, such as `.n2s`, `.n2x`, `.n3`, `.n4`, `.np` and `.nt`.

(102) `tagfeatures([‘n’|_], [cat=n, bar=0]).`

The order of such declarations becomes relevant if one wants to further specify certain but not all sub-categories of a tag class. For reasons of Prolog unification, the declaration for the subclass has to be written first, followed by the more general declaration of the superclass, like in (103).

(103)
```
tagfeatures([‘n’,‘2’|_], [cat=n, bar=0, mass=pl]).
tagfeatures([‘n’|_], [cat=n, bar=0]).
```

12.1.2 `get_tagfeatures/3`

The interface for `tagfeatures/2` in the interface module `linkinfo.pl` is the predicate `get_tagfeatures/3`. It takes a tagged token as its input in the first argument and returns the list of feature-value pairs for the tag of this token in its second and the word form without tag in its third argument. The predicate first splits the tagged token into a list of ASCII numbers for its characters (`atom_chars/2`) and then extracts the characters after the dot, i.e. the tag, as well as the word form from this list (`get_tag/3`). The ASCII list of the tag is transformed back to a list of characters such as described in 12.1.1 above (`make_chars_list/2`).

`get_tagfeatures/3` checks if there is a `tagfeatures/2` declaration in the rule set the first argument of which unifies with the tag character list of the token. It takes the first declaration that unifies. This is why the order of the declarations is relevant: more specific declarations must unify first. If the predicate does not find a unifying declaration in the rule set an empty list is returned, otherwise the feature-value pairs specified in the declaration is delivered.

The predicate is called to separate the word form from its tag in `add_token/3` in the module `linkage2links.pl`.

12.2 Link features

Link types and their subscripts provide information on the type of dependency between the head and the dependent of a link. Whereas link types usually store functional information, subscripts provide additional morphologic or semantic features. A part of this various information is transferred to the realization of a projection type for a link type, the remaining information must be added as feature-value pairs to the AVMs of the head and the dependent of the link.

Link types (Link Grammar connectors) are represented as abbreviations of one or more upper-case letters. Subscripts consist of lower-case letters, each character being one subscript. As certain link types may take multiple subscripts, the rules for link types and subscripts must be declared separately in `linkfeatures.pl`. Otherwise a blow-up of the rule set would be the consequence since the user would have to define a separate rule for every possible combination of link types and subscripts. A subscript, however, can have different meanings depending on the link type it is combined with. Therefore, the user may have to define multiple rules for certain subscripts.

12.2.1 `typefeatures/4`

Link types are defined in the predicate `typefeatures/4` in `linkfeatures.pl`. The arguments of this predicate are (i) the link type, (ii) a list of feature-value pairs to be assigned to the head, (iii) a list of feature-value pairs to be assigned to the dependent, and (iv) the projection type of this link type (cf. 8.1). *EA* links, which connect adjectives to modifying adverbs, can thus be defined as in (104).

(104)
```
typefeatures('EA', [cat=a, bar=0], [cat=adv, bar=0], adjunct).
```

12.2.2 `subsfeatures/4`

The rule set predicate `subsfeatures/4` defines the subscripts to the link types. Its arguments are (i) the subscript, (ii) a list of the link types, in combination with which the current definition of the subscript is applied, (iii) a list of feature-value pairs to be assigned to the head, (iv) a list of feature-value pairs to be assigned to the dependent. *O* links, for instance, connect verbs and their objects, *S* links connect verbs and their subjects. Both link types can be combined with the subscripts for singular and plural, *s* and *p*. If a subject link is said to be singular, the number of both, the verb and the subject, must be singular. However, if an object link is said to be singular, only the number of the object is defined. The verb can be either singular or plural. The user must thus formulate two different declarations for subscript *s*, as shown in (105).

(105)
```
subsfeatures('s', ['O'], [], [num=sg]).
subsfeatures('s', ['S'], [num=sg], [num=sg]).
```

12.2.3 `split_linktype/3`

The interface module `linkinfo.pl` provides two predicates to deal with link type and subscript declarations. `split_linktype/3` takes a link type and its subscripts, and returns the upper-case letter link type in its second and a list of the lower-case letter subscripts in its third argument. This operation is required for the accessing of the separate declarations in the rule set.

12.2.4 `linkfeatures/5`

The predicate `linkfeatures/5` in `linkinfo.pl` delivers (i) the list of feature-value pairs to be assigned to the head, (ii) the list of feature-value pairs to be assigned to the dependent, and (iii) the projection type for a link type and its list of subscripts specified in the first two arguments. It first extracts the feature lists defined for the link type (`get_typefeatures/2`) and then appends the feature lists defined for the subscripts to them (`add_subsfeatures/4`).

12.3 Projection definitions

Several rule set predicates define the way links are converted to particular X-bar structures, i.e. the way their heads are projected. The reader is referred to section 8 for an insight into what these predicates are used for. Most of them have already been mentioned in that section.

12.3.1 `rootlink/2`

Section 8 has shown that one implication of the developed conversion operations is that the linkage to be converted must be a tree structure itself, which again implies that there is a root word in this linkage. The predicate `rootlink/2` defines the link type (and subscripts) the head of which can be the root word of the linkage. Usually, the user may only want complete sentences to be accepted, thus *S* links might be declared to be the only root link type. The user may, however, want the converter to process individual determiner phrases (or noun phrases respectively) if it has not found a whole sentence in the input linkage. The user may thus add a second declaration to the rule set, which allows the converter to do so (106). Note that the order of the declarations is relevant in this case, since the converter is meant to start with a *D* link only if it has not found an *S* link.

(106) `rootlink('S', _AnySubscripts).`
```
rootlink('D', _AnySubscripts).
```

The second argument of the predicate is a list of subscripts to which the root link is restricted. Usually, this feature might not be used, like in the example above. The converter fails if there is no root link defined in the rule set. Note that the predicate `rootlink/2` can only succeed once per linkage. If *D* links were defined as the only root link type, and the input was a whole sentence containing more than one determiner phrases, Link2Tree would only convert the first determiner phrase and omit the rest of the sentence.

The interface predicate to `rootlink/2` in the module `linkinfo.pl` is `is_rootlink/2`, which delivers the two arguments of the former.

12.3.2 `treelevel/5` and `toplevel/1`

The predicate `treelevel/5` defines the projection types and thus the form of X-bar theory to be applied to the conversion. In 8.1, a most basic form of X-bar theory has been met, which is represented in declaration (107).

(107) `treelevel(projection, flat, obligatory, [], [bar=1]).`

The first argument of the predicate defines the name of the projection type. Note that no variables, e.g. `_AnyProjection`, must be used even in this most general case since the variable would be instantiated with the first projection type met in the `typefeatures/4` declarations. Any other projection types would be ignored. It is necessary that the rule set is consistent, i.e. in this case that the projection types used for the definition of the link types are defined explicitly in `treelevel/5`.

The second and third argument of the predicate denote recursiveness and optionality respectively. Note that the values of these arguments are inherent to the program: only the predefined keywords are allowed to be used at these places. The possible values for the second argument are `flat`, `stacked`, `nil`. The third argument can take either `obligatory` or `optional` as its value. Refer to section 8 for a description of the second and the third argument of the predicate.

The forth argument of `treelevel/5` is used to express the ranking of the protection type: it denotes the next lower projection type. All projection type declarations in the rule set form a chain of ranking. A projection type which stands outside this chain is ignored by the converter. The fact that a projection type is part of this chain does not, of course, mean that it must be overtly present in the tree. An optional projection type may not occur in the tree if there is no corresponding link in the input linkage. The lowest projection type takes the empty list as its forth argument. Which of the defined projection type is the topmost projection type is declared separately in the predicate `toplevel/1`.

The features that are added or updated in the AVM of the projection are declared in the last argument of `treelevel/5` as a list of feature-value pairs. Usually, this refers to bar level only.

`get_treelevel/5` and `get_toplevel/1` are the interface predicates to the above predicates in the module `linkinfo.pl`.

12.3.3 `projectable/1`

Each category used in Link2Tree must be defined as projectable or non-projectable. Categories that are not projectable are added as leafs to the tree directly. As by definition they cannot be the head of a larger constituent, links with a non-projectable token as head are omitted. Projectable categories are listed in `projectable/1` facts in the rule set. This special relevance of the category feature induces that `cat` is not only a program inherent keyword but is also compulsory. Tokens that do not have a feature `cat` are treated as non-projectable.

The corresponding interface predicate in `linkinfo.pl` is `is_projectable/1`.

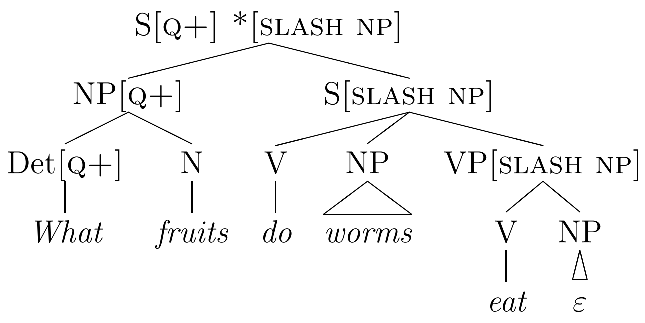
12.3.4 `headfeature/1` and `footfeature/1`

The Head Feature Principle (HFP) and the Foot Feature Principle (FFP) propose that the head features of the head of a tree must be shared with the mother node of that tree, and the foot features of every daughter of that tree must be shared with its mother node. In Link2Tree, the predicates `apply_hfp/2` and `apply_ffp/2` (called from `projection/3` and `project/3`) in the module `merge2tree.pl` are responsible for the application of these principles (cf. section 8.2.5).

Features are defined as head features (and thus transported from head to mother in the tree) in the predicate `headfeature/1`. The interface to it is the predicate `is_headfeature/1` in `linkinfo.pl`. Foot features are defined in `footfeature/1`, the interface of which is `is_footfeature/1`.

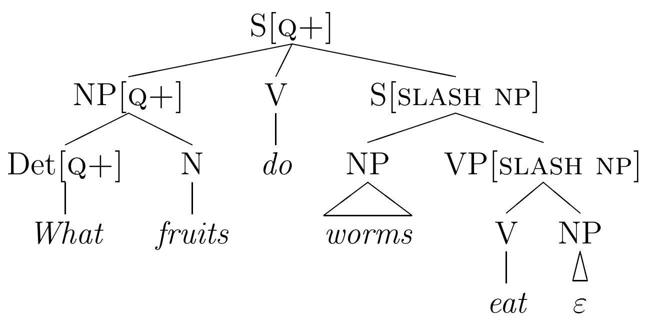
Note that the program inherent features POSITION (9.1.1), HD (7.2) and CAT (8.2.3) must be defined as head features for reasons of proper processing.

The use of foot features, however, has to be considered carefully. As Link2Tree only provides two feature transporting principles, namely HFP and FFP, no further constraints on feature percolation can be made. In (108) two foot features are defined in a GPSG manner: SLASH to denote an absent component and Q to denote a question. SLASH is satisfied if the component specified by it is met as a sister component to the node it is assigned to. Both features are transported up the tree in Link2Tree by application of the FFP. This works fine for Q, which is meant to be transported to the top of the sentence. SLASH, however, is transported further than it should be. In GPSG various feature transporting constraints are defined to prohibit such difficulties.

(108)


In favor of being most generally applicable to various forms of constituency structures, Link2Tree does not define further constraints on feature transport. The idea expressed by SLASH, however, can be realized by traces as demonstrated in 8.3.[24]

If the constituency structure in (108) was slightly changed to the structure given in (109), SLASH could be defined as a head feature and its transport would stop at the right point.

(109)


As a conclusion one can state that in Link2Tree foot features are transported to the root of the tree, whereas head features are only transported to the topmost projection of the node they belong to.

[24]One possibility to stop the transport of a foot feature SLASH at the right point is to define a separate projection type for the link type that connects the head of *what fruits* to the head of *do worms eat*, and to set SLASH on 0 in the definition of that projection type.

13 Program Options

In the current version of Link2Tree three program options can be specified, all of which are concerned with the graphical display of the resulting constituent tree. Options are stored in the argument of the predicate `option/1` in the form `Option=Value`.

The option `bar` indicates if bar level is represented by numbers (`N0, N1, N2`) or by letters (`N, Nbar, NP`). The default value of the option is `0`, which stands for the representation of the bar level by numbers (110).

(110) `option(bar=0).`

The option `display_traces` adds trace indices (cf. 8.3) to the node labels created by `nodelabels/2` in `nicetree.pl`. This option is off by default (value `0`).

The option `display_nodeid` serves to add the identifiers to the nodes in the tree displayed on the screen. This may be useful if one wants to look up the features of particular nodes. The default value of this option is `0`, which means that the node identifiers are not displayed.

Note that all three options take binary values, i.e. either `1` or `0`.

Options are looked up by the predicate `if_option/1` (`linkinfo.pl`). They can be changed by `change_option/1`. If in the argument of this predicate the option name is indicated only, the value of that option is switched. This is meant to work for binary options only. If the argument has the form `Option=Value`, the corresponding option is set to the indicated value. For the altering of their value, the `option/1` facts in the rule set are retracted and asserted again with their new value. The reader is referred to section 6.2 for a description of the user interface for option handling, which makes use of the above predicates.

Options are inherent to the program. They need to be defined in the rule set. Their declarations may not be removed.

14 Relinking rules

Before the actual conversion process is started a preprocessor corrects the input linkage according to rules defined in the rule set. Such rules are called *relinking rules* in Link2Tree. They are stated at the end of the module `linkage2links`. Readers are referred to sections 5.2.2 and 7.3 for a general idea of what relinking does. It has been stated that the user may want to define relinking rules for several reasons: (i) The user may want to correct strange links. Some links or link types returned by Link Grammar differ from the common view of the corresponding syntactic construction. (ii) The user may want to remove irrelevant links (e.g. those to the walls). (iii) The user may want to alter links so that they are equivalent to a specific desired tree output.

The actual point of all the aforementioned reasons is to make the input linkage equivalent to the particular form of constituent structure one desires as an output. A Government & Binding-like tree output, for instance, might require relinking rules different from the ones demanded by a GPSG-like constituent structure. What the equivalence of linkages and constituent trees consists of has been discussed in section 8.1.

Apart from that, some link types need to be corrected for structural reasons. If these link types are left uncorrected the converting algorithm cannot work properly. Link types that violate structural constraints are discussed in 14.2.

14.1 The syntax of relinking rules

At first sight one is tempted to create a single predicate to define relinking rules in a merely declarative way. This is fine for short relinking rules such as "remove links of link type X" or "change the direction of links of link type Y". However, as soon as one wants to define correction rules for more complex situations, the former approach produces Prolog facts of immense size, containing multiple-line arguments. Declarations of such a size have proven to be difficult to understand and extremely error-prone. Finally, one runs into severe problems if one wants to insert new tokens, the positions of which have to be calculated in relation to existing tokens.

What do relinking rules have to be able to express? They must (i) describe conditions under which the rule is applied. These conditions may comprise constraints on the existence or absence of certain links, several of which can be combined, or constraints on the value of features of certain tokens, which again can be combined with other constraints. Relinking rules must define what happens if the conditions are fulfilled: (ii) The removing and (iii) adding of links and tokens, (iv) the asserting of features to the AVMs of certain tokens, (iv) the calculation of the positions of newly added tokens. Finally, relinking syntax needs to (v) provide facilities to find tokens to which links are re-linked.

The chosen syntax for relinking rules is easy to write and read, extendible, and still keeps a declarative character. Relinking rules can be split into a multi-step process: "Check conditions; remove links; insert tokens; add links; add features; a.s.o.". This has been taken into account when the decision about the syntax of relinking rules was made. It is desirable to provide a syntax which is easily understandable and corresponds to the theoretical, i.e. non-Prolog, representation of relinking rules as clearly as possible.

In Link2Tree, relinking rules are represented by the predicate `relink/0`. This predicate is realized by Prolog rules the head of which is `relink:-` and the body of which consists of several terms that describe the constraints under which the rule is applied and what the rule does. These predicates are provided in the module `linkage2links.pl`. They are imported into the rule set, together with `feature/2` from `features.pl`, and make up the syntax for relinking rules.

The module `linkinfo.pl` provides the interface to the relinking rules: the predicate `call_rule/0` (111), which is called from `do_relinking` in the module `linkage2links.pl` (cf. 7.3).

(111)
```
call_rule:-
      relink.
```

14.1.1 Conditions

The first part of a relinking rules describes the condition under which the rule is applied. Three predicates are used to express this condition: `link/4`, `token/2` and `feature/2`. For simple conditions, only one of these may be needed: A rule that removes the irrelevant *AA* links simply takes condition (112). If the user wanted to include empty determiners where there were no overt determiners, one might want to look for nouns that are not head (or dependent, if DP-hypothesis were applied) of a *D* link and neither dependent of an *AN* link, which connects noun-modifiers to following nouns. This would be expressed by condition (113).

(112)
```
link(HeadID, DepID, 'AA', Subscripts)
```

```
(113) token(_Token, TokenID),
      feature(Token, cat=n),
      \+link(TokenID, _, ‘D’, _),
      \+link(_, TokenID, ‘D’, _),
      \+link(_, TokenID, ‘AN’, _)
```

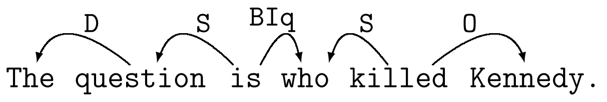
14.1.2 Removing and adding tokens, links and features

The second part of a relinking rule describes the relinking that is done by executing the rule. The module `linkage2links.pl` provides the predicates `remove_link/4` and `add_link/4`, which both take the same arguments as `link/4`, furthermore `remove_token/2` with the arguments of `token/2`, and `add_token/3`, the first argument of which is the token, the second argument a list of feature-value-pairs to be added to its AVM and the third the identifier of its AVM. `add_features/2`, finally, adds a list of feature-value-pairs specified in its second argument to the AVM, the identifier of which is its first argument.

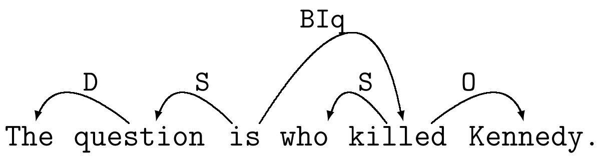
The user may want to have consecutive indices at hand for relinking rules, e.g. for traces. This facility is provided by `new_nr/2`. The predicate `new_nr/2` returns consecutive numbers for the number set specified in its first argument. The user can thus define as many sets of numbers as needed, each of them starting with 0.

14.1.3 Finding remote tokens

Some link types may need to be re-linked to another token, i.e. obtain another head and/or dependent. It is not always trivial to find the needed token since the relation between a known token and the searched token may pass several links. The module `linkage2links.pl` provides the predicates `remote_head/4` and `sub_root/2` to express such far distant relations. These two predicates are discussed by the means of the following example. In linkage (114), the *BIq* link violates the root word constraint: the subordinate clause cannot be reached from the root word *is* in the main clause. One may therefore want to re-link it to *killed*, giving it a new dependent, as shown in (115).

(114) 

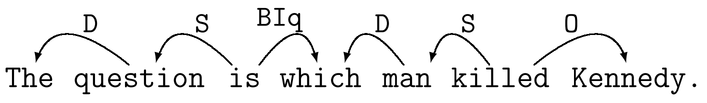
`The question is who killed Kennedy.`

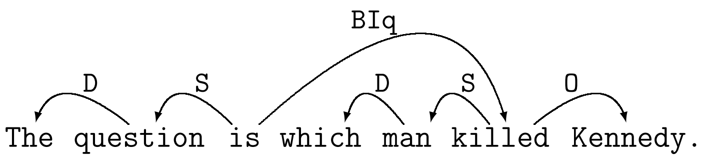
(115) 

The relation between the dependent of the *BIq* link (*who*) and *killed* can simply be expressed by two Prolog facts in a relinking rule (116).

(116)
```
link(Is, Who, 'BI', ['q'|_]),
link(Killed, Who, 'S', _)
```

The searched token (*killed* in our example) may however be further away from the known one (*who*). If the example sentence was *The question is which man killed Kennedy*, this would be the case as shown in linkage (117), which is corrected in linkage (118).

(117) 

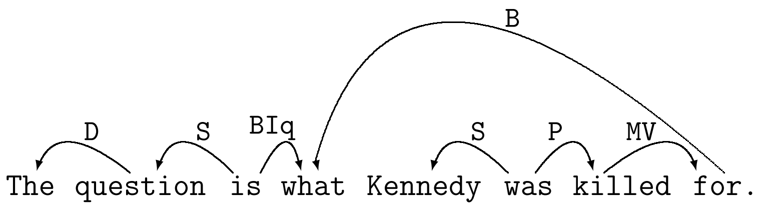
(118) 

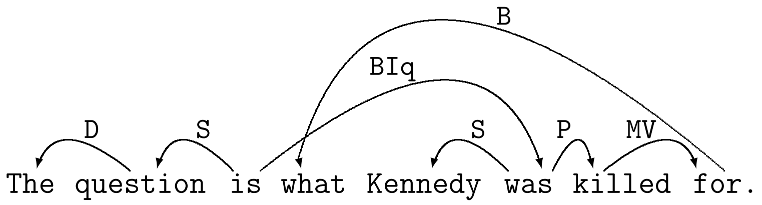
To avoid the necessity to write another rule to transform linkage (117) into linkage (118) the predicate `remote_head/4` is defined. This predicate starts at the token specified in its first argument and moves up the links until it comes to a link of the type and subscripts specified in the second and third argument. `remote_head/4` returns the head of that link. Prolog fact (119) describes the relation between *which* and *killed* in linkage (117) as well as the relation between *who* and *killed* in linkage (114) above.

(119)
```
remote_head(WH, 'S', _AnySubscripts, Killed)
```

The predicate `remote_head/4` can be used as long as one can specify the type and subscripts of the topmost head-link explicitly. However, this may not be useful for the described examples, since the type of the topmost head-link may vary. Linkage (120) shows that not only S but also P (and probably other) links can serve as the topmost head-link to the WH-word in the subordinate clause. Linkage (120) is to be corrected to linkage (121).[25]

[25]Note that the crossing links in linkage (121) indicate that the linkage violates the projectivity constraint. This would result in a wrong word order in the constituent tree – the dominance relations would be correct, though. This is discussed in section 14.2.5.

(120) 

(121) 

Therefore, the predicate `sub_root/2` is provided. It takes a token as its input in the first argument and returns the root word of the sub-linkage that token belongs to. Prolog fact (122) thus describes the relation between the WH-word and the root of the subordinate clause *killed* in all three previous examples (114, 117, 120). Note that the output of the predicate could also be *is*. In the relinking rule, the original *BIq* link would therefore have to be removed before the calling of `sub_root/2` in order to guarantee that only the root of the subordinate clause was returned.

(122) `sub_root(WH, Killed)`

All different cases of *BIq* links can therefore be treated with one single relinking rule (123).

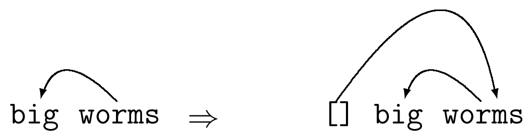
(123)
```
relink:-
    link(BIHead, WH, 'BI', ['q'|BISubs]),
    remove_link(BIHead, WH, 'BI', ['q'|BISubs]),
    sub_root(WH, SubRoot),
    add_link(BIHead, SubRoot, 'BI', ['q'|BISubs]).
```

14.1.4 Calculating new token positions

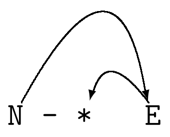
Particularly Government & Binding-like forms of constituency representation cause the user to define relinking rules that insert additional tokens. These tokens are above all non-overt tokens that represent functional categories, such as I (inflection), C (complementizer) or D (determiner). Other possible candidates are empty categories, such as the traces of moved objects in relative clauses or moved auxiliaries in subject-verb inversion. Other than dominance relations, linear precedence is irrelevant for semantic processing of the resulting constituency tree. For an accurate syntactic analysis, however, one may want the inserted token to be at its appropriate position in the sentence. Relinking syntax therefore has to provide instruments to calculate the position of such newly inserted tokens.

If a new token is inserted, there is always a head-dependent relationship to an existing token, to which the new token is going to be linked. Either, the new token is dependent on an existing one, or it is going to be the head of such a token. The position of a new token can thus be calculated in relation to the position of an existing one, which is either its dependent or its head. This is not trivial since there can be further dependents of either the head or the dependent in between.

If one wanted to make the input linkage compatible to DP-hypothesis, an empty determiner would need to be assigned to nouns that have no overt determiner. The inserted determiner would be the head of the existing noun, as demonstrated in (124).

(124)


It can be inferred from example (124) that if the relation of a new token N to an existing token E is that of a head to its dependent, the position of the new token is on the left (or on the right respectively) of the leftmost (or rightmost respectively) dependent of the existing token. This statement is represented by (125).[26]

(125)


The projectivity constraint on linkages (cf. 8.1.1) prevents any dependent of the existing token to stand further left (or right respectively) than the head of this token since the branches in such a tree must not cross.

There are situations other than that described in (124) above, however, where the position of the newly inserted token is to be calculated in relation to its head.

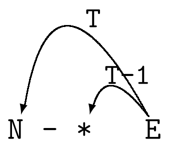
(126) *The apples the worms have eaten were red.*

Sentence (126) is a relative sentence where the object of the subordinate sentence has been moved. The movement is shown in the traced version of the sentence (127).

(127) *[The apples]$_i$ the worms have eaten t_i were red.*

[26]Note that only the left hand variant is shown in this and the following figures, but it always represents the right hand variant too.

We can recognize in (127) that there is an empty object to be inserted, dependent on the verb *eaten*. In this example, thus, an empty token is to be inserted in relation to the position of its head since it has no dependents itself. This causes no problems if the relation of the new token to its existing head is that of a complement to its head, like in (127). But as soon as one wants to insert a specifier to the head, a precise rule is necessary since there may still be complements and adjuncts in between. On the other hand, if complements are inserted, there may still be adjuncts and specifiers around the head and its complement. We are therefore led to figure (128): T denotes the projection type, and T-1 stands for the projection type, which is lower in ranking than T (cf. 8.1.5).

(128)


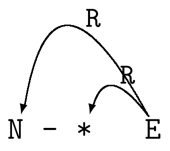
Yet, one still runs into problems if applying (128) to recursive projection types, be they flat or stacked. Sentences (129–131) demonstrate that only the outer object can be moved in a double object construction. Rule (128), however, would insert the empty object inside the existing object, namely outside the utmost dependent of the next lower projection type.

(129) *The book$_i$ I gave her t_i was interesting.*

(130) ** [The girl/She]$_i$ I gave t_i the book was clever.*

(131) *[The girl]$_i$ I gave the book to t_i was clever.*

To handle recursive projection types, an additional rule needs to be defined. We restrict (128) to non-recursive projection types and apply (132) to recursive projection types. The additional rule (132) says that if the relation of a new token N and an existing token E is that of a dependent to its head and if the projection type R of their linking is recursive, the position of the new token is on the left (or right respectively) of the leftmost (or rightmost respectively) dependent of the existing token, the link to which has the same projection type R.

(132)


Figures (125, 128, 132) can be summarized by saying that if one wants to calculate the position of a newly inserted token N, (i) the position of an existing token E that is linked to N, and (ii) the utmost projection type P, which is realized inside E and N, have to be known. The position of the new token is in-between the utmost dependent D of the existing token E which is of the projection type P and the next further existing token F. The position of N is calculated as $\frac{D+F}{2}$. Yet, whether the position of a new token is calculated in relation to its head or to its dependent is to be considered carefully. If the new token has dependents itself it must not be calculated in relation to its head but rather to one of its dependents that immediately follows or precedes it. Otherwise, it will be miss-positioned since its dependents, which may be in between the new token and its head, would not be taken into account.

`linkage2links.pl` provides the predicates `leftendpos/3` and `rightendpos/3`, respectively, to calculate the position of an inserted token. They take the position of the existing token as their first argument and the projection type outside which the link between the new and the existing token is placed as a second. Both predicates return the position of the new token in their third argument.

Assuming the projection types defined in (133), one could thus state relinking rule (134) for assigning an empty determiner to a noun according to DP-hypothesis. As the determiner and the noun are in a head-complement relation, the determiner has to be placed outside the complete sub-linkage of the noun, i.e. outside the realization of the topmost projection type (which is `specifier` in our example).

(133)
```
treelevel(specifier, nil, obligatory, adjunct, [bar=2]).
treelevel(adjunct, stack, optional, complement, [bar=1]).
treelevel(complement, flat, obligatory, [], [bar=1]).
```

(134)
```
relink:-
        % condition
        token(_Noun, NounID),
        feature(NounID, cat=n),
        \+link(NounID, _, 'D', _),
        \+link(_, NounID, 'D', _),
        \+link(_, NounID, 'AN', _),
        % calculation of new token position
        feature(NounID, position=NounPos),
        leftendpos(NounPos, specifier, DetPos),
        % adding of the empty determiner
        add_token([], [cat=d, bar=0, position=DetPos], DetID),
        % relinking
        remove_link(HeadID, NounID, HeadLinktype, HeadSubsList),
        add_link(HeadID, DetID, HeadLinktype, HeadSubsList),
        add_link(DetID, NounID, 'D', HeadSubsList).
```

14.1.5 Summary of the relinking syntax

It has been explained in the current section how relinking rules are to be written and what syntax the module `linkage2links.pl` provides for them. Table 1 summarizes the described predicates.

Conditions:	Adding and removing links:	Auxiliary predicates:
`link/4`	`remove_link/4`	`leftendpos/3`
`token/2`	`add_link/4`	`rightendpos/3`
`feature/2`	`remove_token/2`	`sub_root/2`
	`add_token/3`	`remote_head/4`
	`add_features/2`	`new_nr/2`

Table 1: The syntax of relinking rules

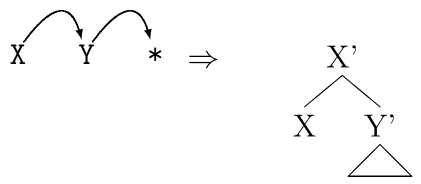
14.2 Link types that violate structural constraints

We have seen that there are several reasons for relinking. Basically, two variants of relinking can be distinguished: relinking that is required for theoretical reasons and relinking required for practical reasons. If relinking is done for theoretical reasons, it is meant to correct a linkage in such a way that the constituent output corresponds to the chosen syntax model. Particular link types may therefore be relinked for the one model of syntactic analysis but kept as they are for the other.

Relinking for practical reasons, however, is necessary in any case. It is required to make the algorithm work properly. Link types that have to be relinked for such reasons violate structural constraints. We have seen in section 8.1.1 that linkages have to fit several constraints to be converted into constituent structures: (i) the linkage has to be directed, (ii) there must be a root word in the linkage, from which every token of the linkage can be reached, (iii) every dependent must only have one head, (iv) the linkage is required to be acyclic, and (v) it has to be projective. Link types that violate one or more of these constraints therefore have to be relinked in order to guarantee the proper working of the conversion algorithm.

The following subsections discuss the different categories of link types that violate structural constraints and how they can be recognized and corrected. To test if a link type violates a structural constraint, the most basic kind of conversion stated in 8.1.1 and in (135), respectively, can be implied.

(135)



The core of the rule set for the above conversion operation consists of a single statement for the ranking of projection types (136) and one single statement for all link types (137).

(136) `treelevel(xbar, stack, optional, [], [bar=1]).`

(137) `typefeatures(_, [], [], xbar).`

Readers are referred to appendix B for the complete rule set.

14.2.1 Link types violating the directionality constraint

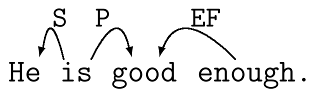
Link Grammar linkages are undirected. In ExtrAns, however, directionality is added to most link types by the predicate `add_dep_info/5`. This predicate is found in the ExtrAns source directory in the file `link_grammar.pl`. Yet, link types that are not explicitly defined as right-head (`h(r)`) or left-head links (`h(l)`) in ExtrAns are assigned no direction (`h(nil)`). These link types therefore violate the directionality constraint. They are arbitrarily changed into left-head links in `linkage2links.pl` (see 7.2.1). The user can easily recognize such link types during the conversion process from the warning message printed on the screen.

Link types violating the directionality constraint comprise *AA, AM, CC, CO, CX, ER, EZ, FL, G, HA, ID, LI, NJ, NT, RW, X* and in some environments *B* (see `link_grammar.pl`). Note that some of these link types still violate another constraint after their conversion to left-head links and must therefore be corrected again in the rule set.

14.2.2 Link types violating the root word constraint

If a link type violates the root word constraint parts of the sentence cannot be reached from the root word because of that link type. If a single token or even a whole part of the input sentence is missing in the output constituent tree, a link type violating the root word constraint must be present in the linkage.

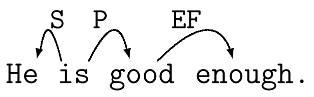
The link type *EF*, for instance, "is used to connect the word 'enough' to adjectives and adverbs." (Temperley et al., 2000). We can see in linkage (138) that the word *enough* cannot be reached from the root word *is*. The tree output for this linkage would therefore be (139), where *enough* is missing.

(138) 

(139)
```
         v1
          |
+---------+
d0        v1
|          |
|     +------+
|     v0     a0
|     |      |
|     |      |
he    is     good
```

Most link types that violate the root word constraint can easily be corrected by changing their direction – some of them need more sophisticated relinking. As soon as relinking rule (140) is added to the rule set in use, the corrected linkage (141) is converted into tree (142).

(140)
```
relink:-
    link(Enough, Adjective, 'EF', EFSubscripts),
    remove_link(Enough, Adjective, 'EF', EFSubscripts),
    add_link(Adjective, Enough, 'EF', EFSubscripts).
```

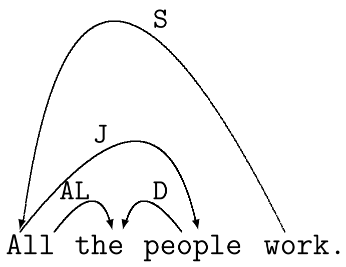
(141) 

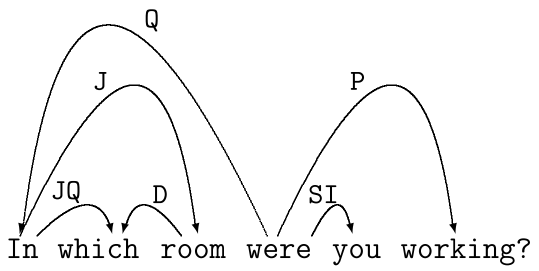
(142)
```
                  v1
                  |
+-----------------+
d0                v1
|                 |
|      +------------+
|      v0           a1
|      |            |
|      |      +--------+
|      |      a0      adv0
|      |      |        |
|      |      |        |
he     is    good    enough
```

Link types violating the root word constraint are *AA, AM, BIq, C, CO, CP, EF, ER, EZ, G, ID, K, L, NI, NJ, NT, OF, PF, QI, U, Qd, Z* and in some environments *QI* and *R* (see 14.2.6). Of these, *AA, AM, CO, EF, EZ, G, K, L, NJ, NT, OF, U, Z* can be corrected simply by changing their direction. The other link types need to be re-linked. The reader is referred to 14.1.3 for a description of the relinking of *BIq* links.

14.2.3 Link types violating the single head constraint

The single head constraint says that a token in a linkage must not have more than one head. Link types that violate the root word constraint usually also violate the single head constraint. Nevertheless, there are link types that violate the single head constraint without violating the root word constraint. These are *AL* and *JQ*. Linkages (143) and (144) show how they are used.

(143) 

(144) 

Linkages that violate the single head constraint without violating the root word constraint have a doubling of words as an effect in the resulting constituent tree. It is the same with cyclic linkages, which are a sub-set of linkages violating the single head constraint (see 14.2.4 below). Tree (145) illustrates how the word *the* is doubled when linkage (143) is converted since it is reached as a dependent from *all* as well as from *people*.

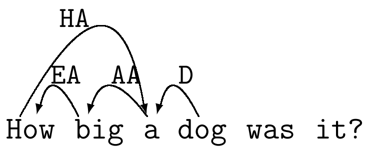
(145)
```
                     v1
                      |
              +--------------+
              d1             v0
              |              |
     +------------+          |
     d1           n1         |
     |            |          |
 +-----+      +------+       |
 d0    d0     d0     n0      |
 |     |      |      |       |
 |     |      |      |       |
all   the    the   people   work
```

AL and *JQ* links can simply be removed as they are redundant because of the existence of *J* links in the respective construction.

14.2.4 Link types violating the circularity constraint

It has already been mentioned above that cyclic linkages are a sub-set of linkages that violate the single head constraint. In Link2Tree, they result in the same error as the latter, namely the doubling of words. Since links are continuously removed during the conversion process cyclic linkages cannot produce a never-ending loop. From a theoretical point of view, however, cyclic links correspond to non-terminating, recursive structures, whereas link types that violate the single head constraint only are equivalent to terminating non-recursive structures.

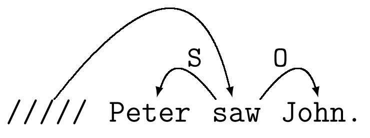
B and *HA* can produce cyclic linkages. They are both arbitrarily changed into left-head links by `linkage2links.pl` as they do not obtain a direction by ExtrAns. Linkage (146) shows the occurrence of *HA* together with *AA*.

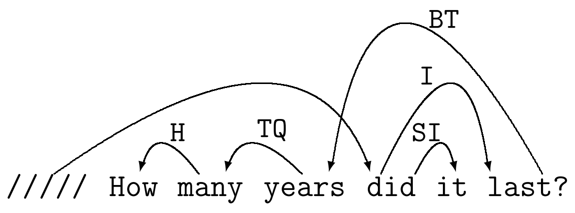
(146)


(146) shows that *HA* is passed repeatedly. *HA* links can simply be removed since every token in the linkage can also be reached without passing the *HA* link. *B* links, on the other hand, need more sophisticated relinking as they occur in various different environments.

14.2.5 Link types violating the projectivity constraint

Link types that violate the projectivity constraint are not easy to recognize in the link structure. However, they cause wrong word order in the constituent tree returned by Link2Tree. To make the violation of projectivity evident in the linkage, an artificial link from the left wall to the root word of the sentence must be added. Linkage (147) does not violate the projectivity constraint, whereas in linkage (148) the *BT* link does so. Tree (149) shows that the output returned for linkage (148) contains wrong word order.

(147)


(148)


(149)

```
                      v1
                      |
    +----------------------------+
    v1                           v1
    |                            |
 +-----+                  +------------+
 v0    d0                 ?1           v0
 |     |                  |            |
 |     |           +-----------+       |
 |     |           d1          ?       |
 |     |           |           |       |
 |     |       +------+        |       |
 |     |       c0     d0       |       |
 |     |       |      |        |       |
 |     |       |      |        |       |
did    it    how    many    years    last
```

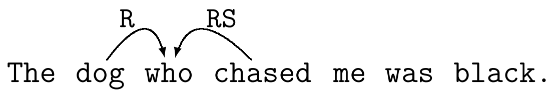
Although the word order in tree (149) is not correct, dominance relations are kept. Such a tree contains crossing branches if one arranges its tokens in the right word order. Depending on what the output of Link2Tree is used for, structures like the one in (149), which are deep-structures (in GB-terminology, see 3.8), may be more suited than any relinked surface-structures since they are much closer to the semantic representation.

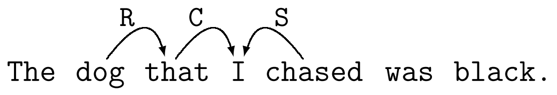
Link types that violate the projectivity constraint comprise *B, BT, CC, ER, Qd, TQ, WR*.

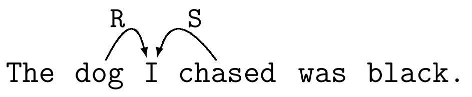
14.2.6 Complex relinking

Not all links behave the same way in all environments. Among those that violate structural constraints under some circumstances but do not violate them under other circumstances are *B, QI, R, W*. Since these link types behave differently in different environments, more than one relinking rule has to be defined for them.

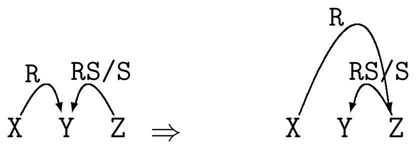
For instance, Temperley et al. (2000) state that "*R* connects nouns to relative clauses. In subject-type relatives, it connects to the relative pronoun [...]; in object-like relatives, it connects either to the relative pronoun or to the subject of the relative clause [...]." These three ways of using *R* are illustrated in linkages (150-152).

(150)


(151)


(152) 

It is evident that *R* violates the root word constraint in linkages (150) and (152), but does not in linkage (151). Therefore, relinking rule (153) for *R* links can be stated, which is only applied if the dependent of the *R* links is at the same time the dependent of either an *S* link or an *RS* link.

(153) 

B links are probably the most complicated links to relink. They usually need to be replaced by trace links. Trace links are not original link types that have to follow structural constraints but are mere auxiliary constructions which are removed and replaced by traces before the actual conversion starts (cf. 8.3). "*B* serves various functions involving relative clauses and questions. It connects transitive verbs back to their objects in relative clauses, questions, and indirect questions [...]; it also connects the main noun to the finite verb in subject-type relative clauses." (Temperley et al., 2000).

Some link types – *Qd, RW, X* – can even be irrelevant under particular circumstances. Their common feature is that they connect to walls or punctuation. They can, however, occur in environments where they cannot just be removed, e.g. link type *W* with coordinating conjunctions.

As already mentioned, more than one relinking rule has to be defined for such link types. Such rules usually involve more than one link type. It is therefore necessary to define rules for particular syntactic constructions rather than for particular link types. The fact that multiple link types are relinked at the same time implies that there is usually more than one way to correct the violation of a structural constraint. Therefore, the relinking of the above link types cannot be done without taking minimal decisions about the model of syntax used. It can be said that relinking is necessary for practical reasons here, but cannot be done without minimal theoretical considerations.

14.2.7 Summary

As demonstrated in this section, some link types need to be relinked for practical reasons. They violate structural constraints which a linkage must comply with in order to be converted properly by Link2Tree. The occurrence of such link types can be recognized in the Link2Tree output. Table 2 summarizes the link types that violate structural constraints.

Violated structural constraint:	Effect in the Link2Tree output:	Link types:
directionality	warning message	*AA, AM, (B), CC, CO, CC, CX, ER, EZ, FL, G, HA, ID, LI, NJ, NT, RW, X*
root word	missing word(s)	*AA, AM, BIq, C, CO, CP, EF, ER, EZ, G, ID, K, L, NI, NJ, NT, OF, PF, Qd, (QI), (R), U, Z*
single head	doubled word(s)	*AL, JQ*
circularity	doubled word(s)	*B, HA*
projectivity	wrong word order	*B, BT, CC, ER, Qd, TQ, WR*

Table 2: Link types that violate structural constraints

15 Conclusion

In this thesis I have presented the dependency-constituency converter Link2Tree. I have described its architecture and functionality in detail. It has been demonstrated what constraints on link structures make them equivalent to constituent structures. This equivalence makes Link2Tree a deterministic converter since a linkage corresponds to exactly one constituent tree. However, the linkages may need some preprocessing, which is called ‘relinking’ in Link2Tree, to ensure their equivalence to a particular form of the constituent structure. A conversion algorithm has been developed and implemented.

Link2Tree is a flexible program, which enables users to specify the form of X-bar theory they desire for the constituent structure delivered by the converter. Furthermore, users can freely choose the features they want to use in the constituent output. The possibilities of Link2Tree make it applicable to Government & Binding as well as PSG structures. If its rule set is tuned accordingly, Link2Tree is able to preserve all information that is stored in linkages for the constituent structures it returns.

At its current state, Link2Tree can only handle simple feature structures. I have shown, however, how facilities to handle complex feature structures can be integrated in Link2Tree in future work.

Part IV

Appendices

A Installation and startup

In order to install Link2Tree first unzip and unpack the file `link2tree.tar.gz` in the extrans source directory, i.e. the directory where the file `main.pl` is (154). Once unpacked, the program consists of the files and directory listed in (155).

(154)
```
~/extrans_1.7> tar xvfz link2tree.tar.gz
```

(155)
```
link2tree/
link2tree/features.pl
link2tree/link2tree.pl
link2tree/linkage2links.pl
link2tree/linkfeatures.pl
link2tree/linkinfo.pl
link2tree/merge2tree.pl
link2tree/nicetree.pl
link2tree/draw.pl
link2tree_start
link2tree_start.pl
```

Link2Tree can be started on its own. To do so, you may have to adopt the paths in the shell-script `link2tree_start` (156) to the paths in the ExtrAns `startup` script.

(156)
```
#!/bin/tcsh -f

setenv JAR "/home/ludwig/rinaldi/WEBEXTRANS/current/JAR"

setenv CLASSPATH ".:$JAR/demo.jar:$JAR/xalan.jar:$JAR/xerces.jar"

echo $CLASSPATH

sicstus -l link2tree_start.pl
```

Shell-script (156) sets the required variables. It then executes the Prolog file `link2tree_start.pl` (157). `link2tree_start.pl` loads ExtrAns by consulting `main.pl` (line 2), as well as the required information for Link Grammar according to the variables set in the shell-script (line 3). The predicate `start_parser/4` (lines 6-7) loads Link Grammar using the parameters specified. At last, `link2tree.pl` is consulted (line 9). Sample queries can now be made by using the predicate `link2tree/3` as demonstrated in (158).

(157)
```
# link2tree_start.pl

:-consult(main).
:-load_foreign. # see link_grammar.pl

# start_parser(Dictionary,Knowledge,Const,Affix)
:-start_parser('link-4.1/data/2.1.dict','link-4.1/data/2.1.knowledge',
               '','link-4.1/data/4.0.affix').

:-consult('link2tree/link2tree.pl').
```

(158)
```
?- link2tree(``cp copies the files.'', 0, Result).
```

B Addendum: `linkfeatures.pl` split up

After the completion of this documentation, I have extended Link2Tree by the facility to load different rule sets while it is running. This enables the user to compare the output of two different rule sets for the same input sentence.

An additional program option has been defined for this in `linkfeatures.pl`: the option `model` specifies the rule set that is to be loaded at the beginning of the conversion process of a sentence. The possible values for `model` can be defined by the user via the predicate `modelpath/2` in `linkfeatures.pl`, which specifies the path of the rule set file, relative to the position from where Link2Tree is started. The auxiliary option `last_model`, which does not need to be altered by the user, memorizes the rule set used for the previous query. If it has not been altered since then, no new rule set needs to be loaded at the beginning of the conversion.

The module `linkinfo.pl` provides a predicate `reload_linkfeatures/0`. This predicate is called at the beginning of `linkage2constituents/4` in `link2tree.pl`, i.e. at the beginning of each conversion. It checks if the option `model` has changed since the last conversion by comparing it to the option `last_model`. If it has changed indeed, `reload_linkfeatures/0` abolishes all predicates imported from the old rule set, loads the new rule set and sets `last_model` to the new rule set.

Listings (B.1) and (B.2) show how the module `linkfeatures.pl` has therefore been split up into a module `linkfeatures.pl` (B.1) and several rule set modules, one of which is shown in listing (B.2).

B.1 Module `linkfeatures.pl`

```
:-module(linkfeatures, [rootlink/2, typefeatures/4, subsfeatures/4,
                        projectable/1, footfeature/1, relink/0,
                        tagfeatures/2, option/1, treelevel/5,
                        toplevel/1, headfeature/1, modelpath/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% linkfeatures.pl
% the rule set module: here, all the information how links shall be
% converted into constituents is stored.
% adaptions need only be made in this module.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% May 2002
% Stefan Hoefler
% shoefler@cl.unizh.ch
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%-------------------------+
% option(OptionName, Value)
%-------------------------+
% Default values for options
% A default has to be defined for every option
% ATTENTION: Do not change Option-Names!

:-dynamic(option/1).
:-dynamic(last_option/1).
option(display_traces=1).
option(display_nodeid=0).
option(bar=0). % 0: n0, n1, n2 // 1: n, nbar, np
option(model=basic).
option(last_model=basic).

% Default rule set (cp. option model)
:-use_module(linkfeatures_basic).

% Paths for rules sets (cp. options model and last_model)
modelpath(basic, 'link2tree/linkfeatures_basic').
modelpath(corr, 'link2tree/linkfeatures_basic_corr').
modelpath(gb, 'link2tree/linkfeatures_gb').

```

B.2 Module `linkfeatures-basic.pl`

```
:-module(linkfeatures_basic, [rootlink/2, typefeatures/4, subsfeatures/4,
                        projectable/1, footfeature/1, relink/0,
                        tagfeatures/2, treelevel/5,
                        toplevel/1, headfeature/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% linkfeatures_basic.pl
% the rule set module: here, all the information how links shall be
% converted into constituents is stored.
% adaptions need only be made in this module.
% NB. this file is the most basic version of a rule set for Link2Tree
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% April 2002
% Stefan Hoefler
% shoefler@cl.unizh.ch
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(linkage2links, [link/4, token/2, add_link/4,
                             remove_link/4, add_token/3,
                             remove_token/2, add_features/2,
                             rightendpos/3, leftendpos/3,
                             remote_head/4, sub_root/2, new_nr/2]).
:-use_module(features, [feature/2]).

%-------------------------------------------------------------------+
% treelevel(TypeOfProjection, Recursive(nil/stack/flat), obligatory/optional,
% Next(Deeper)TypeOfProjection, AdditionalFeaturesListOfTheProjection)
%-------------------------------------------------------------------+

treelevel(xbar, stack, optional, [], [bar=1]).

%-----------------------------+
% toplevel(TopTypeOfProjection)
%-----------------------------+

toplevel(xbar).

%-----------------------------------------------+
% rootlink(RootlinkType, RootlinkSubscriptsList)
%-----------------------------------------------+

rootlink('SFI', _AnySubscripts).
rootlink('SI', _AnySubscripts).
rootlink('SF', _AnySubscripts).
rootlink('S', _AnySubscripts).

%---------------------+
% projectable(Category)
%---------------------+

projectable(_).   % no minors

```

```
%------------------------------------------------+
% typefeatures(LinktypeCapitals, HeadFeatureList,
% DependentFeatureList, ProjectionType)
%------------------------------------------------+

typefeatures(_, [], [], xbar). % no additional information specified

%------------------------------------------------------+
% subsfeatures(+Subscript, +TypeList, -HeadFeatureList,
% -DependendtFeatureList)
%------------------------------------------------------+

subsfeatures(_, [_], [], []). % default

%--------------------+
% headfeature(Feature)
%--------------------+

headfeature(bar).
headfeature(num).
headfeature(rel).

% program inherent features
% do not remove!

headfeature(cat).
headfeature(hd).
headfeature(position). % do not remove this line!
headfeature(trace).    % do not remove this line!


%--------------------+
% footfeature(Feature)
%--------------------+

footfeature(wh).

%------------------------------------------------+
% tagfeatures(TagCharactersList, TagFeautureList)
%------------------------------------------------+
%
```

```
% attention: for affixes that begin with the same letter: write clause
% for the longer affix first
%
% source: extrans_1.7/link-4.1/data21/README_NEWTAGS_WORDS2.1

tagfeatures(['a'|_], [cat=a, bar=0]).
tagfeatures(['c'|_], [cat=c, bar=0]).
tagfeatures(['d'|_], [cat=d, bar=0]).
tagfeatures(['e'|_], [cat=adv, bar=0]).
tagfeatures(['n'|_], [cat=n, bar=0]).
tagfeatures(['o'|_], [cat=p, bar=0]).
tagfeatures(['p'|_], [cat=d, bar=0]).
tagfeatures(['v'|_], [cat=v, bar=0]).
tagfeatures(_,[]). % default: do not delete!

%----------------------------+
% relink
%----------------------------+
%
% ATTENTION: order of relink-rules may be significant

relink. % default. do not remove this line.
```

C Listings of Link2Tree

C.1 Module link2tree.pl

```
:-module(link2tree, [link2tree/3, linkage2constituents/4, chopt/1,
                     lp/3, id/3, feature/3, m/1, test/0, al2t/2]).


%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% link2tree.pl
% converts strings via linkages into constituent trees
% main interface
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% May 2002
% Stefan Hoefler
% shoefler@cl.unizh.ch
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% This version of link2tree combines with ExtrAns 1.7
% (which makes use of Link Grammar 4.1)
% For a detailled description see my master's thesis.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(linkage2links, [linkage2links/2]).
:-use_module(merge2tree, [get_tree_of_links/1]).
:-use_module(nicetree, [wordorder/2, nodelabels/2]).
:-use_module(draw, [draw/1]).
:-use_module(linkinfo, [change_option/1, reload_linkfeatures/0]).
:-use_module(features, [feature/2]).
:-use_module('../link_grammar.pl', [parse/2, print_links/1]).

%-------------------------------------------+
% chopt(+OptionName) or chopt(+Option=Value)
%-------------------------------------------+
% changes program options stored in linkfeatures.pl

chopt(Option):-change_option(Option). % from module linkinfo

m(Model):-chopt(model=Model).

%-------------------------------------------+
% echo_option(?Option=?Value)
%-------------------------------------------+
% displays program options stored in linkfeatures.pl

echo_option(Option=Value):-if_option(Option=Value). % from module


%-----------------------------------------------------+
% link2tree(+String, +ResultLinkageNr, -ConstituentTree)
%-----------------------------------------------------+
% simplified display of results for user queries
% ?- link2tree("cp copies the file.", Tree).

link2tree(String, ResultNr, ConstituentTree):-
        parse(String, Linkages), !,  % from main.pl
        print_links(Linkages), !,  % from main.pl
        nl, nl, write(Linkages), % test!
        linkage2constituents(0, Linkages, ResultNr, ConstituentTree), !,
        display_tree(ConstituentTree),
        !.
%-------------------------------------------------------------+
% linkage2constituents(+SentenceNr, +Linkages, +ResultLinkageNr,
% -ConstituentTree)
%-------------------------------------------------------------+
% interface to other modules/programs
% input:  Link Grammar linkage as returned by ExtrAns 1.7
%         parse/2
% output: constituent tree of the form
%         [NodeNr1,[NodeNr2,'word2'],[NodeNr3, 'word3']]
%         as well as the features to the nodes (dynamically asserted
%         in the file features.pl: feature(NodeNr, Feature=Value).

linkage2constituents(SentNr, Linkages, ResultNr, ConstituentTree):-
        reload_linkfeatures,
        linkage2links(Linkages, ResultNr), !,  % from module linkage2links
        get_tree_of_links(UnarrangedTree), !,  % from module merge2tree
        wordorder(UnarrangedTree, ConstituentTree), !,
                                           % from module nicetree
        nl, nl, write(UnarrangedTree),          % check
        nl, nl, write(ConstituentTree), nl, nl, % check
        postprocess(SentNr, ConstituentTree).

%-----------------------------+
% display_tree(+ConstituentTree)
%-------------------------------+

% display of the resulting constituent tree:
% - lists dynamically asserted features features:feature/2
% - draws tree using draw.pl

display_tree(ConstituentTree):-
        nodelabels(ConstituentTree, LabelledTree), % from module nicetree
        nl, nl, use_draw(LabelledTree),
        nl, nl,
        !.

%==================================================================+
% use_draw
% this predicate makes use of the module draw by Mark Holcomb.
% Draw.pl displays tree-structures in a form similar to
% [s,[np,[d,the],[n,dog]],[vp,[v,ran],[np,[d,the],[n,house]]]].
%
%==================================================================+

%--------------------+
% use_draw(+ListTree)
%--------------------+
% prints an ascii diagram of the tree

use_draw(ListTree):-
        recursive_unif(ListTree, FunctorTree),
        draw(FunctorTree). % from module draw

use_draw(_):-
        nl, nl, write("ERROR in link2tree:use_draw: Tree could not be drawn.").

%----------------------------------------+
% recursive_unif(+ListTree, -FunctorTree)
%----------------------------------------+
% changes a list tree of the form [Node, DaughterTree1, DaughterTree2]
% into a functor tree of the form node(DaughterTree1, Daughtertree2)
% using the predicat unif: Term=..[Functor|Arguments].

recursive_unif([Node|ListRest], FunctorTree):-
```

```
        list_unif(ListRest, FunctorRest),
        FunctorTree=..[Node|FunctorRest],
        !.

recursive_unif(Leaf, Leaf).

%------------------------------------------------------+
% list_unif(+List_of_ListTrees, -List_of_FunctorTrees)
%------------------------------------------------------+

list_unif([], []).

list_unif([FirstListTree|ListRest],[FirstFunctorTree|FunctorRest]):-
        recursive_unif(FirstListTree, FirstFunctorTree),
        list_unif(ListRest, FunctorRest).

%==========================================================+
% postprocess(+SentenceNr, +TreeInListForm)
%==========================================================+
% brings the output into the canonical form we decided on:
% linear precedence: lp(SentNr, LeftSister, RightSister)
% immediate dominance: id(SentNr, Mother, Child)
% features: feature(SentNr, NodeNr, Feature=Value)
%==========================================================+

:-dynamic(id/3).
:-dynamic(lp/3).
:-dynamic(feature/3).

postprocess(SentNr, TreeList):-
        % retracting dynamic facts from the previous query
        my_retractall(id(_,_,_)), % from module linkage2links
        my_retractall(lp(_,_,_)),
        my_retractall(feature(_,_,_)),
        % generating dynamic lp, id and feature facts
        tree2id(SentNr, TreeList),
        tree2lp(SentNr, TreeList),
        add_sentnr_to_features(SentNr),
        % printing the generated facts on the screen
        listing(id/3),
        listing(lp/3),
        listing(feature/3).

%-----------------------------------------------------------+
% tree2id(+SentenceNr, +Tree)
%-----------------------------------------------------------+
% generates the id/3 facts expressing immediate dominance

% case1: one or more complex child(ren)
tree2id(SentNr, [MotherID, [ChildID|Grandchildren]|RestChildren]):-
        assertz(id(SentNr, MotherID, ChildID)),
        tree2id(SentNr, [ChildID|Grandchildren]),
        tree2id(SentNr, [MotherID|RestChildren]).

% case2: leaf
tree2id(_SentNr, [_Terminal, _Leaf]).

%-----------------------------------------------------------+
% tree2lp(+SentenceNr, +Tree)
%-----------------------------------------------------------+
% generates the lp/3 facts expressing linear precedence

% case1: two or more children
tree2lp(S, [M, [N1|C1], [N2|C2]|R]):-
        assertz(lp(S, N1, N2)),
        tree2lp(S, [N1|C1]),
        tree2lp(S, [M, [N2|C2]|R]).

% case2: one complex child
tree2lp(S, [_M, C]):-
        tree2lp(S, C).

% case3: anything else: no child or leaf
tree2lp(_, _).

%-----------------------------------------------------------+
% add_sentnr_to_features(+SentenceNr)
%-----------------------------------------------------------+
% generates feature/3 facts including the sentence number
% (failure-driven loop)

add_sentnr_to_features(SentNr):-
```

```
        feature(Node, Feature=Value), % from module features
        assertz(feature(SentNr, Node, Feature=Value)),
        fail.

add_sentnr_to_features(_).

%============================+
% general (non-iso) predicates
%============================+

%----------------------+
% my_retractall(+Clause)
%----------------------+

my_retractall(Clause):-
        retract(Clause),
        fail.

my_retractall(_).
```

C.2 Module `linkage2links.pl`

```
:-module(linkage2links, [test/0, linkage2links/2,
                         token/2, link/4, add_link/4, remove_link/4,
                         add_token/3, remove_token/2, add_features/2,
                         leftendpos/3, rightendpos/3, remote_head/4,
                         sub_root/2, new_nr/2]).

%==================================================================+
% linkassert: changes the linkage-output of linkgrammar into:
% - token(Word, FeatureMatrixNr)
% - link(HeadNr, DependentNr, Linktype, ListOfSubscripts)
%==================================================================+

:-use_module(features).
:-use_module(linkinfo,[split_linktype/3, get_tagfeatures/3,
                                          call_rule/0, get_treelevel/5]).
:-dynamic(link/4).
:-dynamic(token/2).
:-dynamic(last_tokennr/1).
:-dynamic(last_nr/2).

%------------------------------------------------+
% linkage2links(+Resultlinkages, +ResultLinkageNr)
%------------------------------------------------+

linkage2links(ResultLinkages, ResultNr):-
        retract_all_dynamics,
        get_xth_result(ResultNr, ResultLinkages, Linkage),
        get_lists(Linkage, TokenList, LinkList),
        create_list_of_links(LinkList, ListOfLinks),
        assert_links(ListOfLinks),
        assert_tokens(TokenList),
        do_relinking,
        do_linksorting.

%--------------------+
% retract_all_dynamics
%--------------------+

retract_all_dynamics:-
        my_retractall(link(_,_,_,_)),
        my_retractall(token(_,_)),
        my_retractall(last_tokennr(_)),
        my_retractall(last_nr(_,_)),
        my_retractall(feature(_,_)),   % aus module features
        my_retractall(last_matrixnr(_)). % aus module features

%-------------------------------------------------------+
% get_xth_result(+X, +ResultLinkages, -XthResultLinkage)
%-------------------------------------------------------+

% case1: no result linkages
get_xth_result(_, [], []).

% case2: X is 0 -> get the first result linkage
get_xth_result(0, [FirstLinkage|_RestLinkages], FirstLinkage).

% case3: X > 0 -> check next result linkage and decrease X
get_xth_result(X, [_FirstLinkage|RestLinkages], XthLinkage):-
        X > 0,
        Y is X-1,
        get_xth_result(Y, RestLinkages, XthLinkage).

% case4: X < 0 -> ERROR
get_xth_result(X, _, _):-
        X < 0,
        nl,
        write('ERROR: Number of selected result linkage is negative!'),
        fail.

%-----------------------------------------+
% get_lists(+Linkage, -TokenList, -LinkList)
%-----------------------------------------+

get_lists([[TokenList, LinkList]], TokenList, LinkList).

%---------------------------------------------+
% create_list_of_links(+LinkList, -ListOfLinks)
%---------------------------------------------+

% changes the form of the LinkList from
% [[1,2,h(r),'Ss']] to [link(2,1,'S',[s])]
% in order to simplify assertion of the dynamic link/4 predicate.

create_list_of_links([], []).

create_list_of_links([FirstLink|RestLinks],
                                        [FirstLinkPredicate|RestLinkPredicates]):-
        transform_link(FirstLink, FirstLinkPredicate),
        create_list_of_links(RestLinks, RestLinkPredicates).

%-------------------------------------+
% transform_link(+Link, -Linkpredicate)
%-------------------------------------+

% changes the form of a Link from
% [1,2,h(r),'Ss'] to link(2,1,'S',[s])

% case1: Head is left
transform_link([HeadNr, DependentNr, h(l), Linktype], link(HeadNr,

        split_linktype(Linktype, Type, SubsList). % aus module linkinfo

% case2: Head is right
transform_link([DependentNr, HeadNr, h(r), Linktype], link(HeadNr,

        split_linktype(Linktype, Type, SubsList). % aus module linkinfo

% case3: No Head (no directionality added to this linktype)
% ATTENTION: these linktype are automatically changed into
% left-head-links! this is arbitrary!
% they have to be corrected by an appropriate rule in module linkfeatures!
transform_link([HeadNr, DependentNr, AnyOtherOrNoDirectionality,
                                Linktype], link(HeadNr, DependentNr, Type,
                                                            SubsList)):-
        split_linktype(Linktype, Type, SubsList), % aus module linkinfo
        nl, write('Linktype '), write(Linktype),
        write(' with directionality '), write(AnyOtherOrNoDirectionality),
        write(' arbitrarily changed into left-head-link!'), nl,
        write('Make shure that a correction-rule for this linktype is '),
        write('provided in linkfeatures.pl!'), nl.


```

```
%--------------------------+
% assert_links(+ListOfLinks)
%--------------------------+

% asserts all clauses link/4 of a list

assert_links([]).

assert_links([FirstLinkClause|RestLinkClauses]):-
        assertz(FirstLinkClause),
        nl, write('Link asserted: '), write(FirstLinkClause), nl,
        assert_links(RestLinkClauses).

%----------------------------+
% assert_tokens(+TokenList)
%----------------------------+

assert_tokens([]).

assert_tokens([Token|RestTokens]):-
        new_nr(token, TokenNr),
        add_token(Token, [position=TokenNr, cat='?'], _NewMatrixNr),
        assert_tokens(RestTokens).

%============================+
% general (non-iso) predicates
%============================+

%----------------------+
% my_retractall(+Clause)
%----------------------+

my_retractall(Clause):-
        retract(Clause),
        fail.

my_retractall(_).

%=================+
% relinking process
%=================+

%-------------+
% do_relinking
%-------------+

do_relinking:-
        call_rule, % from module linkinfo
        fail.

do_relinking.

%------------------------------+
% call_list(+ListOfExectuables)
%------------------------------+

call_list([]).

call_list([FirstTerm|RestTerms]):-
        call(FirstTerm),
        call_list(RestTerms).

%===========================+
% predicates for relink-rules
%===========================+
% always use cuts ! in the end of these rules!

%----------------------------------------------+
% remove_link(+HeadNr, +DepNr, +Type, +SubsList)
%----------------------------------------------+

% case1: link exists
remove_link(HeadNr, DepNr, Type, SubsList):-
        retract(link(HeadNr, DepNr, Type, SubsList)),
        nl,
        write('Link removed: '),
        write(link(HeadNr, DepNr, Type, SubsList)),
        nl,
        !.

% cas2: link does not exist
remove_link(HeadNr, DepNr, Type, SubsList):-
```

```
        nl,
        write('ERROR in linkinfo:relink: '),
        write(link(HeadNr, DepNr, Type, SubsList)),
        write(' could not be removed.'),
        nl,
        !.

%------------------------------------------+
% add_link(+HeadNr, +DepNr, +Type, +SubsList)
%------------------------------------------+

add_link(HeadNr, DepNr, Type, SubsList):-
        assertz(link(HeadNr, DepNr, Type, SubsList)),
        nl, write('Link asserted: '),
        write(link(HeadNr, DepNr, Type, SubsList)),
        nl,
        !.

%----------------------------------+
% remove_token(+TokenWord, +MatrixNr)
%----------------------------------+

% case1: token exists
remove_token(Token, Matrix):-
        retract(token(Token, Matrix)),
        my_retractall(feature(Matrix,_)),
        nl, write('Token removed: '),
        write(token(Token, Matrix)),
        nl,
        !.

% case2: token does not exist
remove_token(Token, Matrix):-
        nl,
        write('ERROR in linkinfo:relink: '),
        write(token(Token, Matrix)),
        write(' could not be removed.'),
        nl,
        !.

%--------------------------------------------+
% add_token(+TokenWord, +FeatureList, -MatrixNr)
%-----------------------------------------------+

add_token(Token, FeatureList, MatrixNr):-
        create_matrix(FeatureList, MatrixNr), % from module features
        get_tagfeatures(Token, TagFeatureList, Word), % from module linkinfo
        add_featurelist_to_matrix(MatrixNr, [hd=Word|TagFeatureList]),
                                         % from module features
        assertz(token(Word, MatrixNr)),
        nl, write('Token asserted: '),
        write(token(Word, MatrixNr)),
        nl,
        !.

%-----------------------------------+
% add_features(+Matrix, +FeatureList)
%-----------------------------------+

add_features(Matrix, FeatureList):-
        add_featurelist_to_matrix(Matrix, FeatureList), % from module
                                                        % features
        !.

%---------------------------------------------------+
% remote_head(+Dep, +TopType, +TopSubsList, -TopHead)
%---------------------------------------------------+
%
% finds TopHead-(TopType,TopSubs)->Dep1->Dep2->...->Dep

% case1: TopHead is direct head
remote_head(Dep, TopType, TopSubs, TopHead):-
        link(TopHead, Dep, TopType, TopSubs),
        !.

% case2: recursive
remote_head(Dep, TopType, TopSubs, TopHead):-
        link(Head, Dep, _, _),
        remote_head(Head, TopType, TopSubs, TopHead),
        !.

% case3: no TopHead of TopType/TopSubs exists
```

```
remote_head(_Dep, TopType, TopSubs, _):-
        nl,
        write('ERROR in relinking rule: No TopHead of Type: '),
        write(TopType), write(TopSubs),
        write(' exists. Check rule in module linkfeatures.'),
        nl,!.

%----------------------------------+
% sub_root(+Dependent, -SubRootWord)
%----------------------------------+
%
% finds the root word of a linkage starting from one token

% case1: there is a head
sub_root(Token, RootWord):-
        link(Head, Token, _, _),
        sub_root(Head, RootWord),
        !.

% case2: no head left -> token is root word
sub_root(RootWord, RootWord):-!.

%-----------------------+
% new_nr(+NrType, -NewNr)
%-----------------------+

% case1: last nr of this type existing
new_nr(NrType, NewNr):-
        last_nr(NrType, LastNr),
        NewNr is LastNr +1,
        retract(last_nr(NrType, LastNr)),
        assertz(last_nr(NrType, NewNr)),
        !.

% case2: initialise nr of this type
new_nr(NrType, InitNr):-
        InitNr is 0, % initialise nr
        assertz(last_nr(NrType, InitNr)),
        !.

%==========================================+
% leftendpos(+HeadPos, +ProjType, -LeftEndPos)
%===========================================+

leftendpos(HeadPos, ProjType, LeftEndPos):-
        mindeppos(HeadPos, ProjType, MinDepPos),
        prevpos(MinDepPos, PrevPos),
        LeftEndPos is (MinDepPos+PrevPos)/2,
        !.

%-----------------------------------------+
% mindeppos(+HeadPos, +ProjType, -MinDepPos)
%-----------------------------------------+

% case1: link of this ProjType exists
mindeppos(HeadPos, ProjType, MinDepPos):-
        id2pos(HeadID, HeadPos),
        link(HeadID, DepID, _, ProjType),
        id2pos(DepID, DepPos),
        DepPos<HeadPos,
        smallestdeppos(DepPos, HeadPos, ProjType, MinDepPos),
        !.

% case2: no link of this ProjType exists -> next ProjType
mindeppos(HeadPos, ProjType, MinDepPos):-
        get_treelevel(ProjType, _, _, NextProjType, _),
        mindeppos(HeadPos, NextProjType, MinDepPos),
        !.

% case3: this ProjType is not defined
mindeppos(HeadPos, _ProjType, HeadPos).

%-----------------------------------------------------------+
% smallestdeppos(+StartPos, +HeadPos, +ProjType, -SmallestDepPos)
%-----------------------------------------------------------+

% case1: a link of this ProjType exists
smallestdeppos(StartPos, HeadPos, ProjType, SmallestDepPos):-
        id2pos(HeadID, HeadPos),
        link(HeadID, DepID, _, ProjType),
        id2pos(DepID, DepPos),
        StartPos>DepPos,
        smallestdeppos(DepPos, HeadPos, ProjType, SmallestDepPos),
        !.

% case2: no link of this ProjType with smaller DepPos can be found
smallestdeppos(StartPos, _, _, StartPos).

%-----------------------+
% prevpos(+Pos, -PrevPos)
%-----------------------+

% case1: there is a previous position
prevpos(X, Y):-
        feature(_, position=Y),
        X>Y,
        \+intermediate(X, Y),
        !.

% case2: there is no previous Pos
prevpos(X, Y):-
        Y is X-1.

%===============================================+
% rightendpos(+HeadPos, +ProjType, -RightEndPos)
%===============================================+

rightendpos(HeadPos, ProjType, RightEndPos):-
        maxdeppos(HeadPos, ProjType, MaxDepPos),
        nextpos(MaxDepPos, NextPos),
        RightEndPos is (MaxDepPos+NextPos)/2,
        !.

%-------------------------------------------+
% maxdeppos(+HeadPos, +ProjType, -MaxDepPos)
%-------------------------------------------+

% case1: link of this ProjType exists
maxdeppos(HeadPos, ProjType, MaxDepPos):-
        id2pos(HeadID, HeadPos),
        link(HeadID, DepID, _, ProjType),
        id2pos(DepID, DepPos),
        DepPos>HeadPos,
        biggestdeppos(DepPos, HeadPos, ProjType, MaxDepPos),
        !.

% case2: no link of this ProjType exists -> next ProjType
maxdeppos(HeadPos, ProjType, MaxDepPos):-
        get_treelevel(ProjType, _, _, NextProjType, _),
        maxdeppos(HeadPos, NextProjType, MaxDepPos),
        !.

% case3: this ProjType is not defined
maxdeppos(HeadPos, _ProjType, HeadPos).

%-------------------------------------------------------------+
% biggestdeppos(+StartPos, +HeadPos, +ProjType, -BiggestDepPos)
%-------------------------------------------------------------+

% case1: a link of this ProjType exists
biggestdeppos(StartPos, HeadPos, ProjType, BiggestDepPos):-
        id2pos(HeadID, HeadPos),
        link(HeadID, DepID, _, ProjType),
        id2pos(DepID, DepPos),
        StartPos<DepPos,
        biggestdeppos(DepPos, HeadPos, ProjType, BiggestDepPos),
        !.

% case2: no link of this ProjType with bigger DepPos can be found
biggestdeppos(StartPos, _, _, StartPos).

%------------------------+
% nextpos(+Pos, -NextPos)
%------------------------+

% case1: there is a next position
nextpos(X, Y):-
        feature(_, position=Y),
        X<Y,
        \+intermediate(X, Y),
        !.

% case2: there is no bigger Pos
nextpos(X, Y):-
```

```
        Y is X+1.

%================================================================+
% auxiliary predicates for the calculation of new token positions
%================================================================+

%------------------+
% id2pos(?ID, ?Pos)
%------------------+
id2pos(ID, Pos):-
        feature(ID, position=Pos).

%---------------------+
% intermediate(+X, +Y)
%---------------------+

intermediate(X, Y):-
        feature(_, position=Z),
        X<Z, Z<Y.

intermediate(X, Y):-
        feature(_, position=Z),
        X>Z, Z>Y.

%================================================================%
% do_linksorting/0
%================================================================%
% re-sorts the link/4 facts after relinking according to the order
% given by Link Grammar

do_linksorting:-
        links2list(UnsortedList), !,
        linksort(UnsortedList, SortedList), !,
        list2links(SortedList).

%------------------------+
% links2list(-ListOfLinks)
%------------------------+
% retracts the existing link/4 facts and reads them into a list

links2list([[H, D, T, S]|R]):-
```

```
        retract(link(H, D, T, S)),
        links2list(R).
links2list([]).

%-----------------------------------+
% linksort(+UnsortedList, -SortedList)
%-----------------------------------+
% quicksort for links

linksort([],[]).
linksort([X|Tail], Sorted):-
        linksplit(X, Tail, Small, Big),
        linksort(Small, SortedSmall),
        linksort(Big, SortedBig),
        append(SortedSmall, [X|SortedBig], Sorted).

%---------------------------------+
% linksplit(+X, +Tail, -Small, -Big)
%---------------------------------+
% split for links

linksplit(_X, [], [], []).
linksplit(X, [Y|Tail], [Y|Small], Big):-
        X = [HX, DX|_],
        Y = [HY, DY|_],
        get_left(HX, DX, LX, _RX),
        get_left(HY, DY, LY, _RY),
        LX > LY,
        linksplit(X, Tail, Small, Big).
linksplit(X, [Y|Tail], Small, [Y|Big]):-
        X = [HX, DX|_],
        Y = [HY, DY|_],
        get_left(HX, DX, LX, _RX),
        get_left(HY, DY, LY, _RY),
        LX < LY,
        linksplit(X, Tail, Small, Big).
linksplit(X, [Y|Tail], [Y|Small], Big):-
        X = [HX, DX|_],
        Y = [HY, DY|_],
        get_left(HX, DX, LX, RX),
        get_left(HY, DY, LY, RY),
        LX = LY,
        RX < RY,
        linksplit(X, Tail, Small, Big).
linksplit(X, [Y|Tail], Small, [Y|Big]):-
        X = [HX, DX|_],
        Y = [HY, DY|_],
        get_left(HX, DX, LX, _RX),
        get_left(HY, DY, LY, _RY),
        LX = LY,
        linksplit(X, Tail, Small, Big).

%--------------------------------------------------------+
% get_left(+Nr1, +Nr2, -SmallerPosition, -BiggerPosition)
%--------------------------------------------------------+

get_left(A, B, L, R):-
        feature(A, position=APos),
        feature(B, position=BPos),
        APos < BPos,
        L = APos,
        R = BPos, !.
get_left(A, B, L, R):-
        feature(A, position=APos),
        feature(B, position=BPos),
        L = BPos,
        R = APos, !.

%-----------------------+
% list2links(+ListOfLinks)
%-----------------------+
% asserts link/4 facts from a list of links

list2links([[H, D, T, S]|R]):-
        assertz(link(H, D, T, S)),
        list2links(R).
list2links([]).

%----------+
% append/3
%----------+

append([], List, List).
append([Element], List, [Element|List]).
append([Element|Rest], List, [Element|AppendList]):-
        append(Rest, List, AppendList).
```

C.3 Module merge2tree.pl

```
:-module(merge2tree, [get_tree_of_links/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% merge2tree.pl - converts links into a tree
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(linkinfo, [get_toplevel/1, get_treelevel/5,
                        is_projectable/1, is_rootlink/2,
                        is_footfeature/1, linkfeatures/5,
                        is_headfeature/1]).
:-use_module(linkage2links, [new_nr/2, link/4, token/2]).
:-use_module(features).

%--------------------------+
% get_tree_of_links(-Tree)
%--------------------------+
% gets the root link type and checks if a link of this type
% exists. The head of this link is the starting point of the
% conversion process.

get_tree_of_links(Tree):-
        get_rootlink(RootType, RootSubsList),
        link(StartHeadNr, _DependentNr, RootType, RootSubsList),
                                       % from module linkage2links
        merge2tree(StartHeadNr, Tree).

%----------------------------+
% merge2tree(+HeadNr, -Tree)
%----------------------------+

% case1: the category of the head is projectable
% - gets the top projection type (eg. specifier) (get_toplevel/1)
% - resolves (retracts) all links from this head of the projection type "trace" and
% adds a trace index to their tokens (trace/1).
% - starts the projection of a tree with HeadNr as its head


merge2tree(HeadNr, Tree):-
        cat_is_projectable(HeadNr),
        get_toplevel(ProjType), % from module linkinfo
        trace(HeadNr),
        projection(ProjType, HeadNr, Tree),
        !.

% case2: the category of the head is not projectable
% - turn the head into a terminal subtree

merge2tree(HeadNr, LeafTree):-
        add_leaf(HeadNr, LeafTree).

%---------------+
% trace(+HeadNr)
%---------------+
% resolves (retracts) all links from this head of the projection type "trace" and
% initialises and adds trace indices to the feature structure of their
% tokens (trace/1).

% case1: if trace exists
trace(HeadNr):-
        link(HeadNr, TraceNr, Type, SubsList),
        linkfeatures(Type, SubsList, HeadFeatures, TraceFeatures, trace),
                                % from module linkinfo
        add_features_to_both(HeadNr, HeadFeatures, TraceNr, TraceFeatures),
        new_nr(trace, TraceIndex), % from module linkage2links
        add_features_to_both(HeadNr, [trace=TraceIndex], TraceNr,
                        [trace=TraceIndex]),
        retract(link(HeadNr, TraceNr, Type, SubsList)),
        trace(HeadNr).

% case2: no trace exists
trace(_HeadNr).

%-------------------------------------------------+
% projection(+ProjectionType, +HeadNr, -ProjectedTree)
%-------------------------------------------------+

% case1: does a link from this head and of this projection type exist?
% yes. -> add the features defined for its link type to the feature
% structures of its head and its dependents; retract the link; project
% the head to a new node along the projection mode defined for this
% projection type; convert the dependent and all its dependents
% (recursively) to a subtree; check if this projection type is
% recursive.

projection(ProjType, HeadNr, [ProjNr, [DepProjNr|DepTree]|RestTree]):-
        link(HeadNr, DepNr, Type, SubsList),
        linkfeatures(Type, SubsList, HeadFeatures, DepFeatures, ProjType),
        add_features_to_both(HeadNr, HeadFeatures, DepNr, DepFeatures),
        retract(link(HeadNr, DepNr, Type, SubsList)),
        project(HeadNr, ProjType, ProjNr),
        merge2tree(DepNr, [DepProjNr|DepTree]),
        apply_ffp(DepProjNr, ProjNr),
        recursive_projection(ProjType, HeadNr, ProjNr, RestTree).

% case2: does a link from this head and of this projection type exist?
% no. -> Is this projection type obligatory? yes. -> project its head
% (whithout any dependents) to a new node along the projection mode
% defined for this projection type; go to the links of the next (lower)
% projection type

projection(ProjType, HeadNr, [ProjNr, NextSubTree]):-
        get_treelevel(ProjType, _Recursive, obligatory, _NextProjType,
                  _ProjFeatures),
        project(HeadNr, ProjType, ProjNr),
        next_projection(ProjType, HeadNr, NextSubTree).

% case2: does a link from this head and of this projection type exist?
% no. -> Is this projection type obligatory? no. -> go to the links of
% the next (lower) projection type.

projection(ProjType, HeadNr, NextSubTree):-
        next_projection(ProjType, HeadNr, NextSubTree).


%---------------------------------------------+
% project(+HeadNr, +ProjType, -ProjNr)
%---------------------------------------------+
% projects the token and creates a new avm for its projection
%- get the features which have to be added (updated) to the projection
%(eg. bar=1)
% - create a new matrix
% - add the headfeatures to the new matrix
% - add the footfeatures to the new matrix
% - add (update) the features from above to the avm of the projection
% - return the ID of the projection

project(HeadNr, ProjType, ProjNr):-
        get_treelevel(ProjType, _Recursive, _Obligatory, _NextProjType,
                                 ProjFeatures),
        create_matrix([], ProjNr),
        apply_hfp(HeadNr, ProjNr),
        apply_ffp(HeadNr, ProjNr),
        add_featurelist_to_matrix(ProjNr, ProjFeatures).

%-------------------------------------------------------------+
% recursive_projection(+ProjType, +HeadNr, +ProjNr, -ProjTree
%-------------------------------------------------------------+

% case1: ProjType produces stacked structure if multiple links occur
% Does another link of this head and projection type exist && is this
% projection type recursive? yes and it produces stacked
% structures. -> ...
recursive_projection(ProjType, HeadNr, _ProjNr, [ProjTree]):-
        link(HeadNr, _DepNr, Type, SubsList),
        linkfeatures(Type, SubsList, _HeadFeatures, _DepFeatures, ProjType),
        get_treelevel(ProjType, stack, _Obligatory, _NextProjType,
                                 _ProjFeatures),
        projection(ProjType, HeadNr, ProjTree).

% case 2: ProjType produces flat structure if multiple links occur
recursive_projection(ProjType, HeadNr, HeadProjNr, [[DepProjNr|DepTree]|RestTree]):-
        link(HeadNr, DepNr, Type, SubsList),
        linkfeatures(Type, SubsList, HeadFeatures, DepFeatures, ProjType),
        get_treelevel(ProjType, flat, _Obligatory, _NextProjType,
                                 _ProjFeatures),
        add_features_to_both(HeadNr, HeadFeatures, DepNr, DepFeatures),
        retract(link(HeadNr, DepNr, Type, SubsList)),
        merge2tree(DepNr, [DepProjNr|DepTree]),
        apply_ffp(DepProjNr, HeadProjNr),
        recursive_projection(ProjType, HeadNr, HeadProjNr, RestTree).

% case3: ProjType is not recursive or no links of this type are left
recursive_projection(ProjType, HeadNr, _ProjNr, [NextSubTree]):-
```

```
        next_projection(ProjType, HeadNr,  NextSubTree).

%--------------------------------------------------+
% next_projection(+PrevProjType, +HeadNr, -ProjTree)
%--------------------------------------------------+

% case2: lowest projection type is reached
next_projection(ProjType, HeadNr, TerminalSubTree):-
        get_treelevel(ProjType, _Recursive, _Obligatory, [], _ProjFeatures),
        add_leaf(HeadNr, TerminalSubTree).

% case1: there is a next projection type left
next_projection(ProjType, HeadNr, ProjTree):-
        get_treelevel(ProjType, _Recursive, _Obligatory, NextProjType,
                                  _ProjFeatures),
        projection(NextProjType, HeadNr, ProjTree).

%-----------------------------------+
% add_leaf(+LeafNr, -TerminalSubTree)
%-----------------------------------+

add_leaf(LeafNr, [LeafNr, LeafWord]):-
        trace(LeafNr),
        token(LeafWord, LeafNr).

%-------------------------------------------------------------------------------+
% add_features_to_both(+HeadNr, +HeadFeautures, +DependentNr, +DependentFeatures)
%-------------------------------------------------------------------------------+

add_features_to_both(HeadNr, HeadFeatures, DependentNr, DependentFeatures):-
        add_featurelist_to_matrix(HeadNr, HeadFeatures), % from module features
        add_featurelist_to_matrix(DependentNr, DependentFeatures). % from features

%----------------------------+
% cat_is_projectable(+TokenNr)
%----------------------------+

% if feature cat is already part of the matrix
cat_is_projectable(TokenNr):-
        feature(TokenNr, cat=Cat),
        is_projectable(Cat).  % aus module linkinfo
```

```

cat_is_projectable(HeadNr):-   % Token ist Head
        link(HeadNr, _DepNr, Type, SubsList),
        linkfeatures(Type, SubsList, HeadFeatures, _DepFeatures,
                                 _Projectiontype),
        get_feature(HeadFeatures, cat=Cat),   % from module features
        is_projectable(Cat).   % from module linkinfo

%---------------------------------------------------+
% get_rootlink(-RootlinkType, -RootlinkSubscriptsList)
%---------------------------------------------------+

get_rootlink(RootType, RootSubsList):-
        link(_Head, _Dependent, RootType, RootSubsList),
                                         % from module linkages2links
        is_rootlink(RootType, RootSubsList). % from module linkinfo

get_rootlink(_Rootlink, _RootSubsList):- % error message, if no root link exists
        nl, write('ERROR in linkinfo:get_rootlink/2: no rootlink found.'),
        nl, write('Execution aborted.'),
        fail.

%---------------------------------+
% apply_ffp(+DepNr, +ProjectionNr)
%---------------------------------+

apply_ffp(DepNr, ProjectionNr):- % failure-driven loop
        feature(DepNr, Feature=Value), % from module features
        is_footfeature(Feature), % from module linkinfo
        update_feature(ProjectionNr, Feature=Value),
        fail.

apply_ffp(_DepNr, _ProjectionNr).

%----------------------------------+
% apply_hfp(+HeadNr, +ProjectionNr)
%----------------------------------+

apply_hfp(HeadNr, ProjectionNr):- % failure-driven loop
        feature(HeadNr, Feature=Value), % aus module features
        is_headfeature(Feature), % aus module linkinfo
        update_feature(ProjectionNr, Feature=Value),
        fail.

apply_hfp(_HeadNr, _ProjectionNr).
```

C.4 Module nicetree.pl

```
:-module(nicetree, [wordorder/2, nodelabels/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% nicetree.pl
% - rearrenges a tree according to its word order
% - labels tree
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(features, [feature/2]).
:-use_module(linkinfo, [if_option/1]).

%----------------------------------------------+
% wordorder(+RandomlyArrangedTree, -ArrangedTree)
%----------------------------------------------+

% case0: empty tree
wordorder([], []).

% case1: tree of one or more elements
wordorder([MatrixNr|DaughterTrees],
                  [MatrixNr|RecursivelySortedDaughters]):-
        quicksort(DaughterTrees, SortedDaughterTrees),
        recursivesort(SortedDaughterTrees, RecursivelySortedDaughters).

wordorder(Element, Element).

%----------------------------------------------------+
% recursivesort(+List_of_Trees, -List_of_sorted_Trees)
%----------------------------------------------------+

recursivesort([FirstTree|RestTrees],
                           [SortedFirstTree|SortedRestTrees]):-
        wordorder(FirstTree, SortedFirstTree),
        recursivesort(RestTrees, SortedRestTrees).

recursivesort([], []).

%-----------------------+
% quicksort for MatrixNrs
%-----------------------+

quicksort([],[]).

quicksort([X|Tail], Sorted) :-
    split(X, Tail, Small, Big),
    quicksort(Small, SortedSmall),
    quicksort(Big, SortedBig),
    append(SortedSmall, [X|SortedBig], Sorted).

split(_X, [],[],[]).
split(X, [Y|Tail], [Y|Small], Big) :-
    % X > Y, !,
        X=[XID|_], Y=[YID|_],
        feature(XID, position=PositionX),
        feature(YID, position=PositionY),
        PositionX > PositionY, !,
    split(X, Tail, Small, Big).
split(X, [Y|Tail], Small, [Y|Big]) :-
    split(X, Tail, Small, Big).

%-------------------------------------------+
% nodelabels(+UnlabelledTree, -LabelledTree)
%-------------------------------------------+

nodelabels([], []).

nodelabels([MatrixNr|Daughters], [Label|LabeledDaughters]):-
        matrixnr2label(MatrixNr, Label),
        recursivelabels(Daughters, LabeledDaughters).

nodelabels(Leaf, Leaf).

%---------------------------------------------------+
% recursivelabels(+ListofTrees, -ListofLabeledTrees)
%---------------------------------------------------+

recursivelabels([FirstTree|RestTrees],
                                 [LabeledFirstTree|LabeledRestTrees]):-
        nodelabels(FirstTree, LabeledFirstTree),
        recursivelabels(RestTrees, LabeledRestTrees).

recursivelabels([], []).

%---------------------------------+
% matrixnr2label(+MatrixNr, -Label)
%---------------------------------+

matrixnr2label(MatrixNr, Label):-
        add_catbar2label(MatrixNr, CatBarLabel),
        add_trace2label(MatrixNr, CatBarLabel, TracedLabel),
        add_matrixnr2label(MatrixNr, TracedLabel, Label).

%-----------------------------------+
% add_catbar2label(+MatrixNr, -Label)
%-----------------------------------+

add_catbar2label(MatrixNr, Label):-
        feature(MatrixNr, cat=Cat), % wenn cat vorhanden
        name(Cat, CatList),
        feature(MatrixNr, bar=BarNr), % wenn bar vorhanden
        barnr2barletter(BarNr, BarLetter), % n2 -> np etc.
        name(BarLetter, BarList),
        append(CatList, BarList, LabelList),
        name(Label, LabelList).

add_catbar2label(MatrixNr, Label):-
        feature(MatrixNr, cat=Label). % wenn nur cat (ohne bar) vorhanden

add_catbar2label(_MatrixNr, ''). % wenn weder cat noch bar vorhanden

%-----------------------------------+
% barnr2barletter(+BarNr, -BarLetter)
%-----------------------------------+

barnr2barletter(BarNr, BarLetter):-
        if_option(bar=1), % from module linkinfo
        bar(BarNr, BarLetter).

barnr2barletter(BarNr, BarNr).

%---------------------+
% bar(BarNr, BarLetter)
```

```
%--------------------+

bar(0,'').
bar(1,'bar').
bar(2, 'p').

%--------------------------------------------+
% add_trace2label(+Matrix, +Label, -TracedLabel)
%--------------------------------------------+

% case1: if trace exists and trace-display-option on
add_trace2label(Matrix, Label, TracedLabel):-
        if_option(display_traces=1), % aus module linkinfo
        feature(Matrix, trace=TraceNr),
        atom_chars(Label, LabelList1),
        name(TraceNr, TraceNrList),
        atom_chars('.t', PreTraceList),
        atom_chars('', PostTraceList),
        append(LabelList1, PreTraceList, LabelList2),
        append(LabelList2, TraceNrList, LabelList3),
        append(LabelList3, PostTraceList, LabelList4),
        atom_chars(TracedLabel, LabelList4).

% case2: if no trace exists
add_trace2label(_Matrix, Label, Label).

%-----------------------------------------------------+
% add_matrixnr2label(+Matrix, +Label, -LabelInclMatrixNr)
%-----------------------------------------------------+

add_matrixnr2label(Matrix, Label, CompleteLabel):-
        if_option(display_nodeid=1), % aus module linkinfo
        atom_chars(Label, LabelList),
        name(Matrix, MatrixList),
        atom_chars('<', OpenBrList),
        atom_chars('>', CloseBrList),
        append(OpenBrList, MatrixList, LabelList1),
        append(LabelList1, CloseBrList, LabelList2),
        append(LabelList2, LabelList, LabelList4),
        atom_chars(CompleteLabel, LabelList4).

```

```
add_matrixnr2label(_Matrix, Label, Label).

%=====================+
% auxiliary predicates
%=====================+

%-------------------------------------+
% append(+List1, +List2, -CombinedList
%-------------------------------------+

append([], List, List).
append([Element], List, [Element|List]).
append([Element|Rest], List, [Element|AppendList]):-
        append(Rest, List, AppendList).

```

C.5 Module features.pl

```
:-module(features, [feature/2, add_featurelist_to_matrix/2,
                    create_matrix/2, update_feature/2,
                    get_feature/2, last_matrixnr/1]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% feature.pl - erstellt und bearbeitet Feature-Matricen:
% - feature(MatrixNr, Feature=Value)
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(linkinfo, [is_headfeature/1]).

:-dynamic(last_matrixnr/1).
:-dynamic(feature/2).


%--------------------------------+
% make_new_matrixnr(-NewMatrixNr)
%--------------------------------+

make_new_matrixnr(NewNr):-   % wenn schon eine LastNr vorhanden
        last_matrixnr(LastNr),
        NewNr is LastNr +1,
        retract(last_matrixnr(LastNr)),
        assertz(last_matrixnr(NewNr)),
        !.

make_new_matrixnr(0):-   % MatrixNr initialisieren
        assertz(last_matrixnr(0)).

%---------------------------------------+
% create_matrix(+FeatureList, -MatrixNr)
%---------------------------------------+

create_matrix(FeatureList, MatrixNr):-
        make_new_matrixnr(MatrixNr),
        add_featurelist_to_matrix(MatrixNr, FeatureList).

%---------------------------------------------------+
% add_featurelist_to_matrix(+MatrixNr, +FeatureList)
%---------------------------------------------------+

add_featurelist_to_matrix(_MatrixNr, []).
add_featurelist_to_matrix(MatrixNr,
                          [FirstFeature=FirstValue|RestFeatureList]):-
        feature(MatrixNr, FirstFeature=_),   % Feature vorhanden
        update_feature(MatrixNr, FirstFeature=FirstValue),
        add_featurelist_to_matrix(MatrixNr, RestFeatureList).

add_featurelist_to_matrix(MatrixNr,
                          [FirstFeature|RestFeatureList]):-
        assertz(feature(MatrixNr, FirstFeature)),   % neues Feature
        add_featurelist_to_matrix(MatrixNr, RestFeatureList).

%-----------------------------------------------+
% update_feature(+MatrixNr, +Feature=NewValue)
%-----------------------------------------------+

update_feature(MatrixNr, Feature=NewValue):-   % feature vorhanden
        retract(feature(MatrixNr, Feature=_OldValue)),
        assertz(feature(MatrixNr, Feature=NewValue)).

update_feature(MatrixNr, Feature=Value):-   % zur Sicherheit: neues
                                            % Feature
        assertz(feature(MatrixNr, Feature=Value)).

%---------------------------------------------+
% get_feature(+FeatureList, ?Feature=?Value)
%---------------------------------------------+
% Fail if feature or value resp. not available

get_feature([Feature=Value|_], Feature=Value).

get_feature([_|Rest], Feature=Value):-
        get_feature(Rest, Feature=Value).
```

C.6 Module `linkinfo.pl`

```
:-module(linkinfo, [linkfeatures/5, is_rootlink/2, is_projectable/1,
                    is_footfeature/1, split_linktype/3, call_rule/0,
                    get_tagfeatures/3, change_option/1, if_option/1,
                    get_treelevel/5, get_toplevel/1,
                    is_headfeature/1, reload_linkfeatures/0]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% linkinfo.pl
% interface between the "grammar" (module linkfeatures) and the
% converter.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% April 2002
% Stefan Hoefler
% shoefler@cl.unizh.ch
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

:-use_module(linkfeatures).

:-dynamic(linkfeatures/4).

%-----------------------------------------------+
% linkfeatures(+Type, +SubsList, -HeadFeatureList,
% -DependentFeatureList, -ProjectionType)
%-----------------------------------------------+

linkfeatures(Type, SubsList, HeadFeatureList, DependentFeatureList,
                         ProjectionType):-
        get_typefeatures(Type, HeadFeatureList_1, DepFeatureList_1,
                                         ProjectionType),
        add_subsfeatures(SubsList, Type, HeadFeatureList_1,
                                         DepFeatureList_1, HeadFeatureList,
                                         DependentFeatureList),
        !.

%---------------------------------------------------------------+
% get_typefeatures(+Type, -HeadFeatureList, -DependentFeatureList,
% -ProjectionType)
%---------------------------------------------------------------+

get_typefeatures(Type, HeadFeatureList, DepFeatureList,
                                 ProjectionType):-
        typefeatures(Type, HeadFeatureList, DepFeatureList,
                                 ProjectionType).

%-------------------------------------------------+
% split_linktype(+Linktype, -Type, -SubscriptList)
%-------------------------------------------------+

split_linktype(Linktype, Type, SubsList):-
        name(Linktype, CharList),
        get_capitals(CharList, CapitalsList, MinorsList),
        name(Type, CapitalsList),
        name_subscripts(MinorsList, SubsList).

%---------------------------------------------------+
% name_subscripts(+MinorsASCIIList, -SubscriptsList)
%---------------------------------------------------+

name_subscripts([], []).

name_subscripts([FirstASCII|RestASCII],[FirstSub|RestSubs]):-
        atom_chars(FirstSub, [FirstASCII]),
        name_subscripts(RestASCII, RestSubs).

%------------------------------------------------------+
% get_capitals(+CharList, -CapitalsList, -RestCharList)
%------------------------------------------------------+

% case1: empty character list
get_capitals([], [], []).  % capitals aus einer leeren Liste

% case2: first element is a capital
get_capitals([FirstChar|RestChars], [FirstChar|RestCapitals],
                         RestCharList):-
        65 =< FirstChar, FirstChar =< 90,
        get_capitals(RestChars, RestCapitals, RestCharList),
        !.

% case3: first element is $
get_capitals([FirstChar|RestChars], [FirstChar|RestCapitals],
                         RestCharList):-
        36=:=FirstChar,
        get_capitals(RestChars, RestCapitals, RestCharList),
        !.

% case4: first element is no capital (nor $)
get_capitals(RestChars, [], RestChars).  % subscripts

%-----------------------------------------------------------+
% add_subsfeatures(+SubscriptsList, +Type, +HeadFeatureList,
% +DependentFeatureList, -CompleteHeadFeautureList,
% -CompleteDependentFeatureList)
%-----------------------------------------------------------+

% if no subscritps left
add_subsfeatures([], _Type, HeadFeatureList, DepFeatureList,
                 HeadFeatureList, DepFeatureList).

% there are subscripts left
add_subsfeatures([Subscript|RestSubs], Type, HeadFeatureList_1,
                 DepFeatureList_1, HeadFeatureList, DepFeatureList):-
        get_subsfeatures(Subscript, Type, SubsHeadFeatureList, SubsDepFeatureList),
        append_featurelist(HeadFeatureList_1, SubsHeadFeatureList,
                                          HeadFeatureList_2),
        append_featurelist(DepFeatureList_1, SubsDepFeatureList,
                                          DepFeatureList_2),
        add_subsfeatures(RestSubs, Type, HeadFeatureList_2, DepFeatureList_2,
                                         HeadFeatureList, DepFeatureList).

%----------------------------------------------------------+
% get_subsfeatures(+Subscript, +Type, -SubsHeadFeatureList,
% -SubsDepFeatureList)
%----------------------------------------------------------+

get_subsfeatures(Subscript, Type, SubsHeadFeatureList,
                                SubsDepFeatureList):-
        subsfeatures(Subscript, TypeList, SubsHeadFeatureList,
                                SubsDepFeatureList),
        element_of_list(Type, TypeList).

get_subsfeatures(_Subscript, _Type, [], []).  % unknown subscripts

```

```
%---------------------------------+
% element_of_list(+Element, +List)
%---------------------------------+

element_of_list(Element, [Element|_Rest]).
element_of_list(Element, [_FirstElement|Rest]):-
        element_of_list(Element, Rest).

%--------------------------------------------------+
% append_featurelist(+ListA, +ListB, -CombinedList)
%--------------------------------------------------+

append_featurelist([], List, List).
append_featurelist([Element], List, [Element|List]).
append_featurelist([First|Rest], List, [First|AppendList]):-
        append_featurelist(Rest, List, AppendList).

%---------------------+
% is_projectable(?Cat)
%---------------------+

is_projectable(Cat):-
        projectable(Cat).   % aus module linkfeatures

%-----------------------------------------------+
% is_rootlink(+RootlinkType, +RootSubscriptsLink)
%-----------------------------------------------+

is_rootlink(RootType, RootSubsList):- % wenn ein Rootlink vorhanden
        rootlink(RootType, RootSubsList). % aus module linkfeatures

%-------------------------+
% is_footfeature(?Feature)
%-------------------------+

is_footfeature(Feature):-
        footfeature(Feature).   % aus module linkfeatures

%-------------------------+
% is_headfeature(?Feature)
%-------------------------+

is_headfeature(Feature):-
        headfeature(Feature).   % aus module linkfeatures

%----------------------------+
% call_rule
%----------------------------+
% interface to module linkfeatures

call_rule:-
        relink. % from module linkfeatures

%------------------------+
% get_treelevel(_,_,_,_,_)
%------------------------+
% interface to module linkfeatures

get_treelevel(ProjType, Recursive, Obligatory, NextProjType, ProjFeatures):-
        treelevel(ProjType, Recursive, Obligatory, NextProjType, ProjFeatures).

%-------------------------+
% get_toplevel(TopProjType)
%-------------------------+

get_toplevel(TopProjType):-
        toplevel(TopProjType).

%=================+
% handling options
%=================+

%------------------------------------------------------------+
% change_option(+OptionName) or change_option(+OptionName=Value)
%------------------------------------------------------------+

% case1: Option=Value, Option already exists
change_option(Option=NewValue):-
        option(Option=OldValue), % from module linkfeatures
        retract(option(Option=OldValue)),
        assertz(option(Option=NewValue)),
        nl, write('Option '), write(Option), write(' set to '),
        write(NewValue), nl.

% case2: only Option indicated, Option set to 0
change_option(Option):-
        option(Option=0), % from module linkfeatures
        retract(option(Option=0)),
        assertz(option(Option=1)),
        nl, write('Option '), write(Option), write(' set to 1'), nl.

% case3: only Option indicated, Option set to 1
change_option(Option):-
        option(Option=1), % from module linkfeatures
        retract(option(Option=1)),
        assertz(option(Option=0)),
        nl, write('Option '), write(Option), write(' set to 0'), nl.

% case4: error in option handling
change_option(_):-
        nl,
        write('ERROR: Option does not exist or is not used in this way.'),
        nl, write('Check module linkfeatures.'), nl.

%---------------------------+
% if_option(+Option=Value)
%---------------------------+

if_option(Option=Value):-
        option(Option=Value). % from module linkfeatures

%==================================================+
% get the features of tokens out of its affixed tag
%==================================================+

%-------------------------------------------------+
% get_tagfeatures(+Token, -TagFeaturesList, -Word)
%-------------------------------------------------+

get_tagfeatures(Token, TagFeatures, Word):-
        atom_chars(Token, CharList),
        check4features(CharList, TagFeatures, Word),
        !.
```

```

%---------------------------------------------------+
% check4features(+ASCIIList, -TagFeauturesList, -Word)
%---------------------------------------------------+

check4features(TokenASCIIList, TagFeatures, Word):-
        get_tag(TokenASCIIList, TagASCIIList, WordASCIIList),
        make_chars_list(TagASCIIList, TagCharsList),
        atom_chars(Word, WordASCIIList),
        tagfeatures(TagCharsList, TagFeatures). % from module linkfeatures

check4features(_, []).

%----------------------------------------------------+
% get_tag(+WordASCIIList, -TagASCIIList, -WordformList)
%----------------------------------------------------+

% case1: empty ASCII-List
get_tag([],[], []).

% case2a: a full stop is the only symbol
get_tag([FullStop], [], [FullStop]):-
        FullStop=:=46.

% case2b: first symbol is a dot -> rest is tag
get_tag([FirstASCII|Tag],Tag, []):-
        FirstASCII=:=46.

% case3: cut first symbol
get_tag([FirstASCII|RestASCII],Tag, [FirstASCII|RestWordform]):-
        get_tag(RestASCII, Tag, RestWordform).


%---------------------------------------+
% make_chars_list(+ASCIIList, -CharsList)
%---------------------------------------+

make_chars_list([],[]).

make_chars_list([ASCII|RestASCII],[Char|RestChars]):-
        atom_chars(Char, [ASCII]),
        make_chars_list(RestASCII, RestChars).

%==================================================================+
% reload_linkfeatures/0: load a another rule set during the program
%==================================================================+

% case1: the option model has not been changed since the last time
reload_linkfeatures:-
        option(last_model=Model),
        option(model=Model).

% case2: the option model has been changed since the last time
reload_linkfeatures:-
        abolish([rootlink/2, typefeatures/4, subsfeatures/4,
             projectable/1, footfeature/1, relink/0, tagfeatures/2,
             treelevel/5, toplevel/1, headfeature/1]),
        option(model=Model),
        modelpath(Model, ModelPath),
        use_module(ModelPath, [rootlink/2, typefeatures/4, subsfeatures/4,
                          projectable/1, footfeature/1, relink/0,
                          tagfeatures/2, treelevel/5, toplevel/1,
                          headfeature/1]),
        change_option(last_model=Model).

```

C.7 Module linkfeatures.pl

```
:-module(linkfeatures, [rootlink/2, typefeatures/4, subsfeatures/4,
                        projectable/1, footfeature/1, relink/0,
                        tagfeatures/2, option/1, treelevel/5,
                        toplevel/1, headfeature/1, modelpath/2]).

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% linkfeatures.pl
% the rule set module: here, all the information how links shall be
% converted into constituents is stored.
% adaptions need only be made in this module.
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
% May 2002
% Stefan Hoefler
% shoefler@cl.unizh.ch
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%--------------------------+
% option(OptionName, Value)
%--------------------------+
% Default values for options
% A default has to be defined for every option
% ATTENTION: Do not change Option-Names!

:-dynamic(option/1).
:-dynamic(last_option/1).
option(display_traces=1).
option(display_nodeid=0).
option(bar=0). % 0: n0, n1, n2 // 1: n, nbar, np
option(model=basic).
option(last_model=basic).

% Default rule set (cp. option model)
:-use_module(linkfeatures_basic).

% Paths for rules sets (cp. options model and last_model)
modelpath(basic, 'link2tree/linkfeatures_basic').
modelpath(corr, 'link2tree/linkfeatures_basic_corr').
modelpath(gb, 'link2tree/linkfeatures_gb').

```

References

Bennett, Paul. 1995. *A Course in Generalized Phrase Structure Grammar*. London: UCL Press.

Bloomfield, Leonard. 1933. *Language*. New York: H. Holt and Company.

Borsley, Robert D. 1997. *Syntax Theorie: Ein zusammengefasster Zugang*. Tübingen: Niemeyer Verlag.

Bröker, Norbert and Geert-Jan Kruijff. 1999. "Dependency Grammar." Homepage at the Institute of Formal and Applied Linguistics, Prague University, Prague. http://ufal.mff.cuni.cz/dg/.

Burton-Roberts, Noel. 1997. *Analyzing Sentences*. 2nd ed. New York: Addison Wesley Longman.

Chomsky, Noam. 1957. *Syntactic Structure*. The Hague: Mouton.

Chomsky, Noam. 1981. *Lectures on Government and Binding*. 7th ed. Berlin and New York: Mouton de Gruyter.

Chomsky, Noam. 1986a. *Barriers*. Cambridge, MA: MIT Press.

Chomsky, Noam. 1986b. *Knowledge of Language: Its Nature, Origin and Use*. New York: Praeger.

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

Cook, Vivian and Mark Newson. 1996. *Chomsky's Universal Grammar: An Introduction*. 2nd ed. Oxford: Blackwell.

Covington, Michael. 1992. "GB Theory as Dependency Grammar." Research Report AI-1992-03, University of Georgia, Athens GE.

Covington, Michael. 1994. "An Empirically Motivated Reinterpretation of Dependency Grammar." Research Report AI-1994-01, University of Georgia, Athens GE.

Fraser, Norman. 1996. "Dependency Grammar." In: Brown, Keith and Jim Miller (eds.), *Concise Encyclopedia of Syntactic Theories*. Oxford: Elsevier, 71–75.

Gaifman, Haim. 1965. "Dependency systems and phrase-structure systems." *Information and Control* vol. 8, 304–337.

Gazdar, Gerald, Ewan Klein, Geoffrey Pullum and Ivan Sag. 1985. *Generalized Phrase Structure Grammar*. Cambridge, MA: Harvard University Press.

Gazdar, Gerald and Chris Mellish. 1989. *Natural Language Processing in PROLOG*. Workingham, Reading MA: Addison-Wesley.

Haegeman, Liliane. 1994. *Introduction to Government & Binding Theory*. 2nd ed. Oxford: Blackwell.

Hays, David G. 1964. "Dependency theory: A formalism and some observations." *Language* vol. 40, 511–525.

Hudson, Richard. 1990. *English Word Grammar*. Oxford: Basil Blackwell.

Järvinen, Timo and Pasi Tapanainen. 1997. "A Dependency Parser for English." Department of Linguistics Technical Report TR-1, University of Helsinki, Helsinki.

Matthews, Peter H. 1981. *Syntax*. Cambridge: Cambridge University Press.

Meĺčuk, Igor A. 1988. *Dependency Syntax: Theory and Practice*. New York: State University of New York Press.

Mollá, Diego. 2000. *ExtrAns: An Answer Extraction System for Unix Manpages: On-line Manual*. University of Zurich: Computional Linguistics, Zurich.

Mollá, Diego, Gerold Schneider, Rolf Schwitter and Michael Hess. 2000a. "Answer Extraction Using a Dependency Grammar in ExtrAns." *T.A.L.* vol. 41, no. 1, 127–156.

Mollá, Diego, Rolf Schwitter, Michael Hess and Rachel Fournier. 2000b. "ExtrAns, an Answer Extraction System." *T.A.L.* vol. 41, no. 2, 1–25.

Müller, Stefan. 1998. "Prolog und Computerlinguistik: Teil I Syntax." Vorlesungsskripte Computerlinguistik, Humboldt Universität, Berlin. http://www.dfki.de/∼stefan/Pub/prolog.html.

Pollard, Carl and Ivan Sag. 1987. *Fundamentals*, vol. 1 of *Information-based Syntax and Semantics*. Stanford: University of Chicago Press.

Pollard, Carl and Ivan Sag. 1994. *Head-Driven Phrase Structure Grammar*. Chicago: University of Chicago Press.

Radford, Andrew. 1997. *Syntactic theory and the structure of English: A minimalist approach*. Cambridge: Cambridge University Press.

Sag, Ivan A. and Thomas Wasow. 1999. *Syntactic Theory: A Formal Introduction*. Stanford, California: CSLI.

Schneider, Gerold. 1998a. "An Introduction to Government & Binding: Notes for an Imaginary Colloquium." unpublished paper, University of Zurich.

Schneider, Gerold. 1998b. "A Linguistic Comparison of Constituency, Dependency and Link Grammar." Master's thesis, Philosophical Faculty I of the University of Zurich, Zurich.

Schneider, Gerold. 1999. "Wordtags as used for the ExtrAns version of Link Grammar Lexicon V 2.1." Available together with the ExtrAns system in the file README_NEWTAGS_WORDS2.1 in the ExtrAns parser directory.

Sleator, Daniel and Davy Temperley. 1993. "Parsing English with a Link Grammar." Proceedings of IWPT '93.

Somers, Harold. 1984. "On the validity of the complement-adjunct distinction in valency grammar." *Linguistics* vol. 22, 507–530.

Tarvainen, Kalevi. 1981. *Einführung in die Dependenzgrammatik*. Reihe Germanistische Linguistik 35, Tübingen: Niemeyer.

Temperley, Davy, Daniel Sleator and John Lafferty. 2000. "Link Grammar Documentation." Homepage at the Carnegie Mellon University. http://bobo.link.cs.cmu.edu/grammar/html/dict/.

Tesnière, Lucien. 1959. *Eléments de syntaxe structurale*. Paris: Librairie Klincksieck.

Weber, Heinz. 1997. *Dependenzgrammatik*. Tübingen: Gunter Narr Verlag.

Index