Generating Multilingual Natural Language Expressions for Grail Concepts

Part I: Theory

Wim Claassen
University of Nijmegen
NICI

Contents

1 Introduction 1
1.1 Grail 1
1.2 Why NLG? 2
1.3 How could this work? 2
2 Syntactic issues in Multilingual natural language generation 3
2.1 Generic linguistic framework 3
2.2 Language Specific components 5
2.2.1 Grammar specification 5
2.2.2 Populating the Lexicon 6
2.3 Syntactic tree formation in Segment Grammar 6
2.3.1 Syntactic trees 6
2.3.2 Word order 10
2.3.3 Word form 11
3 Generating phrases to describe Grail concepts 11
3.1 Annotating grail concepts with linguistic entities 12
3.1.1 Concept annotations 12
3.1.2 Relation annotations and syntactic frames. 12
3.2 Modeling schemes and filtering 15
3.2.1 Feature-State scheme 15
3.2.2 Surgical Procedures 16
3.2.3 Unwanted criteria 17
3.2.4 Tagging and filtering 17
3.3 The Natural Language Generation algorithm 18
3.3.1 Generating the constituent structure 19
3.3.2 Serializing the constituent structure 20
3.3.3 Producing the surface string 21
4 References 21

1 Introduction

The documentation on the ROIS based Natural Language Generator developed within the context of the Galen In Use project consist of three papers: (1) a theoretical paper on the design of the generator, (2) a description of its implementation, and (3) a manual describing how the generator is to be used.
The current paper provides some theoretical background on the Multilingual Natural Language Generator that is under development at NICI within the context of the Galen In Use project. The goal is to develop a multilingual natural language generator that produces natural language descriptions of conceptual structures in a concept representation language called Grail. The generator is based on Segment Grammar, and implemented using the knowledge representation tool ROIS (Relation Oriented Inference System). The generator consists of a generic algorithm (implemented in ROIS NPL) that is applied to language specific lexicons and grammars. The key design requirements are flexibility, extendibility, usefulness of the generator in practical applications, and the possibility to reuse the language specific lexicons and grammars in a multilingual natural language analyzer. The initial target languages are Dutch, English and Finnish.
This paper assumes a basic understanding of the medical concept representation language Grail. If you are unfamiliar with notions such as ‘subsumption hierarchy’, ‘prototypes’ and ‘defining criteria’ please consult some introductory material on Grail. In addition we assume that you are familiar with ROIS and the ROIS development tools Idefix and Mole. In order to be able to use the natural language generation software package to create a generator for a particular language you will also have to be familiar with both the basic linguistic notions concerning syntactic structures in general and the actual grammar rules of that language. The rest of this section will provide a high-level overview of the problems, and the solutions adopted.

1.1 Grail

The input structures to the generator are Grail concepts such as (1) and (2) below[1].

```
(1)  (Fracture which < hasLocation Femur >)
(2)  (SurgicalDeed which <
       isCharacterisedBy
          (performance whichG < isEnactmentOf
             (Removal which <
               actsSpecificallyOn
                     (Abscess which < hasLocation ExternalEar
                                      hasSeverity (Severity which <
                                                       hasAbsoluteState severe >) >)
               hasExtend complete >) >)
       isCharacterisedBy
          (nonPerformance whichG < isEnactmentOf
             (Incision which < actsSpecificallyOn Pinna >) >) >)
```

Concept (1) could be described in English by e.g., the phrases *fracture of the femur* or *femoral fracture* and concept (2) by the phrase *removal of severe abscess of external ear without incision of auricle*.

An important aspect of the formal concept representation language Grail is that it is compositional. Primitive concepts, such as *Fracture* and *Femur*, can be combined into composite Grail concepts[2] such as *(Fracture which < hasLocation Femur >)*. A knowledge engineer defines the primitive concepts of interest and a set of rules that specify how concepts can be combined into more complex composite

[1] For matters of convenience Grail concepts are often presented by lines of Grail source code that could be evaluated by a Grail source code compiler such as the GCE Workspace. Note however that the written presentation of the canonical form of the resulting concept will often differ more or less substantially from the source code presentation. The canonical presentation of example (2) for instance, will not contain the concept SurgicalDeed, as in the current CORE model SurgicalDeed is an alias of the composite concept (Process which < hasClinicalRole SurgicalRole >). Instead the primitive concept Process and its defining criterion hasClinicalRole-SurgicalRole will appear.

In the rest of this paper composite Grail concepts are presented in canonical form. The syntax of the canonical presentation is similar to Grail, only the semantics differ. The major reason for using the canonical presentation is that it immediately shows what the base and the defining criteria of the composite concept are.

[2] Composite Grail concepts are sometimes called *prototypes* or *particularisations*. I prefer to use the less technical term *composite concepts*.

concepts. A given set of primitive concepts and combination rules defines a so-called *model of concepts*, which is the set of all concepts that can be described using the primitive concepts and the combination rules. In this sense Grail is said to be a *generative* concept representation language.

1.2 Why NLG?

In essence Grail is intended as a conceptual interlingua between medical applications and coding schemes in the medical domain. However, although concepts from a Grail model can be presented visually to humans by expressions in the Grail language, structures such as (2) above clearly illustrate that non-Grail experts will need a more comprehensible presentations such as a natural language phrases.

If all primitive and composite concepts of a Grail model could be expressed using single content words (e.g., the noun *fracture*), producing multilingual natural language descriptions for Grail concepts would be a rather trivial and uninteresting table look-up task. The problem becomes interesting however, when single words are not sufficient to describe a composite Grail concept. In that case it is necessary to describe the concept using a more complex natural language phrase, possibly containing prepositional phrases, adjectives and other modifiers.

1.3 How could this work?

The hypothesis is that a natural language description of a Grail concept can be produced by combining the words that express the individual components of the concept in a syntactically correct way. For example: *(Fracture which < hasLocation Femur >)* can be described by the English noun phrases *fracture of the femur* or *femoral fracture*. These two phrases are alternative natural language descriptions that combine the English words *femur, femoral* and *fracture* using a prepositional phrase and an adjectival phrase respectively.

Now in order to be able to produce sensible natural language descriptions of Grail concepts we have to annotate[3] the individual concepts of a Grail model with the individual lemmas of the natural language, and in addition we have to annotate the concept combination rules of a Grail model (represented by so-called *statements*) to the phrase and word combination rules (the grammar) of the natural language.

The example above requires e.g., that the Grail concept *Fracture* maps to the English noun *fracture* and the Grail concept *Femur* is annotated with the English noun *femur*. Next to these concept annotations the example above requires that the Grail combination rule *X hasLocation Y* is annotated with "prepositional phrase + preposition *of*" such that the phrase that expresses *X* (*fracture*) is modified by a prepositional phrase that expresses *Y* (*of the femur*).

Note that this rule does not imply that e.g., *(Femur which isLocationOf Fracture)* can be expressed as *femur of the fracture*. Instead, to describe this example concept properly would require an additional annotation that maps *X which isLocationOf Y* to a noun phrase describing *X* (*femur*) which should be modified by an adjectival phrase describing *Y* (*fractured*) to produce the more acceptable *fractured femur*.

In order to provide for a multilingual generator that is easy to extend and to maintain we chose to design a generic (language independent) generation algorithm that applies language specific grammars and lexicons to produce utterances in multiple languages. Within the Galen In Use project the emphasis is on the generation of noun phrases but we felt that the generator should also be able to produce complete sentences. We chose Segment Grammar (Kempen, 1987; Kempen ea 1987) as the linguistic framework, because it has shown to be well-suited for both natural language generation and interpretation.

The rest of this paper is structured as follows: section 2 focuses on syntactic issues of the natural language generation process. Semantic issues are dealt with in section 3 where we describe the way Grail concepts are 'translated' into phrases. Then section 0 describes how to produce the language specific data required for a particular language (grammar rules, lemmas/word forms required and semantic mappings). In section 0 we describe how you can use the development and maintenance tools to use these

[3] Traditionally, within the GALEN and GALEN-IN-USE projects the term *linguistic annotations* is used to refer to the semantic mappings between a Grail model and a natural language.

data to build your own language and semantic models. Finally section 0 provides a detailed description of the implementation of the natural language generation modules.

2 *Syntactic issues in Multilingual natural language generation*

Strictly speaking, the term ‘multilingual’ applies to any natural language generator that is able to produce phrases in at least two natural languages. However, in my opinion a truly multilingual generator should consist of a single generic (language independent) component and language specific components for each of the individual languages covered. A major design goal of the present project is to limit the language specific components of the generator to the language specific data that reside in the lexicon and the grammar. As a consequence the generator should have no need for any language specific algorithms, and extending the generator to cover an extra language would only require the specification of the grammar and a lexicon.

Although the focus of the present paper is on generation, the next sections are also applicable to the natural language analysis process. The application of generic multilingual generation and analysis algorithms requires a generic linguistic framework that can support the complete range of syntactic phenomena that are present in each of the individual language fragments to be covered. This framework is described below in implementation independent terms. The actual ROIS implementation of the framework is presented in section 3.1.

2.1 Generic linguistic framework

This section goes into the entities and relations that form the building blocks of the Generic Linguistic Framework. Why and how these objects are used will be described in the sections that follow. The objects used within the generic linguistic framework are defined in table 1. We distinguish eight elementary entities and one composite entitity. The elementary entities are: *lemma categories*, *features*, *feature values*, *lexeme categories*, *lexemes*, *spellings*, *phrase categories*, *syntactic functions*, *positions* and *segments*. Please observe the following naming conventions: the names of lemma categories, phrase categories, and positions start with a capital letter. Lemma- and phrase categories can also be referred to by their abbreviations which are presented in the table within parentheses. The names of the other elementary entities start with a small letter.

Feature-value pairs are composed of a feature and a value. They are represented by joining together the feature and the value using an equals sign (=). For example, *number=singular*. Segments are composite entities which are defined by a triple consisting of a phrase category, a syntactic function, and a phrase- or lemma category. A segment’s name is formed by joining together the names of its elements using hyphens as indicated in the table. The role of segments will be described in more detail in section 2.3.

lemma category	*Noun* (*N*) | *Adjective* (*ADJ*) | *Article* (*ART*) | *Preposition* (*PREP*) | *Adverb* (*ADV*) | *ProperName* (*PN*) | *CoordinatingConjunction* (*COOCON*) | *SubordinatingConjunction* (*SUBCON*) | *MainVerb* (*MV*) | *AuxiliaryVerb* (*AV*) | *CopulaVerb* (*CV*) | *CardinalNumber* (*CARD*) | *PersonalPronoun* (*PERSPRO*) | *PossessivePronoun* (*POSSPRO)* | *DemonstrativePronoun* (*DEMONPRO*) | *InterrogativePronoun* (*INTERPRO)* | *IndefinitePronoun* (*INDEFPRO*) | *ReflexivePronoun* (REFLPRO) | *ReciprocalPronoun* (*RECIPRO*) | *RelativePronoun* (*RELPRO*)
feature	*number* | *gender* | *definite* | *case* | *prenominal* | *inflection* | *affixRole* | *countable* | *determinable* | *tense* | *aspect* | *participle* | *syntacticallyTransitive* | *syntacticallyReflexive* | *reciprocal* | *separableVerb* | *diminutiveForm*
feature value	*singular* | *plural* | *masculine* | *feminine* | *neuter* | + *(positive)* | - *(negative)* | *nominative* | *genitive* | *dative* | *accusative* | *translative* | *partitive* | *essive* | *inessive* | *adessive* | *illative* | *allative* | *elative* | *ablative* | *instructive* | *abessive* | *prefix* | *infix* | *suffix* | *past* | *present* | *future* | *perfect* | *imperfect* | *presentParticiple* | *pastParticiple*
lexeme category	<string> (e.g., *basic noun, uninflected adjective, plural noun*)
lexeme	<spelling>
spelling	<string>
phrase category	*NounPhrase* (*NP*) | *PrepositionalNounPhrase* (*PNP)* | *AdjectivalPhrase* (*ADJP*) | *AdverbialPhrase* (*ADVP*) | *Sentence* (*S*)
syntactic function	*head* | *modifier* | *functor* | *determiner* | *prefix* | *postfix* | *subject* | *directObject* | *indirectObject* | *complement* | *auxiliary* | *particle* | *predicate* | *conjunctionElement*

position	*1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9*
feature-value pair	*<feature>, = , <value>*
segment	*<phrase category>, -, <syntactic function>,-,<phrase category> | <lemma category>*

table 1 Elementary en composite entities within the generic linguistic framework

The entities of the linguistic framework are involved in the following relations (table 2):

lemma category	*feature*	*n, n*
lemma	*lemma category*	*n, 1*
lexeme category	*lemma category*	*n, 1*
lexeme category	*feature-value pair*	*n, n*
lexeme	*lemma*	*n, 1*
lexeme	*lexeme category*	*n, 1*
lexeme	*spelling*	*n, 1*
phrase category	*feature*	*n, n*
segment	*feature*	*n, n*
segment	*position*	*n, 1*
number	*singular* | *plural*	*n, 1*
gender	*masculine* | *feminine* | *neuter* | *nonNeuter*	*n, 1*
definite	+ | -	*n, 1*
case	*nominative* | *genitive* | *dative* | *accusative* | *translative* | *partitive* | *essive* | *inessive* | *adessive* | *illative* | *allative* | *elative* | *ablative* | *instructive* | *abessive*	*n, 1*
prenominal	+	*n, 1*
inflection	+ | -	*n, 1*
affixRole	*prefix* | *infix* | *suffix*	*n, 1*
countable	+ | -	*n, 1*
determinable	+ | -	*n, 1*
tense	*past* | *present* | *future*	*n, 1*
aspect	*perfect* | *imperfect*	*n, 1*
participle	*presentParticiple* | *pastParticiple*	*n, 1*
syntacticallyTransitive	+ | -	*n, 1*
syntacticallyReflexive	+ | -	*n, 1*
reciprocal	+ | -	*n, 1*
separableVerb	+ | -	*n, 1*
diminutiveForm	+ | -	*n, 1*

table 2. Possible relations between syntactic entities and corresponding cardinality values

Informally the contents of the table is described as follows: lemma categories (e.g., *noun)* have zero or more features (e.g. *number*). Lemmas (e.g., *liver*) have exactly one lemma category (e.g. *noun*). Lexeme categories (e.g., *plural noun*) have exactly one lemma category (e.g., *noun*) and zero or more feature-value pairs (e.g., *number=plural*). Lexemes (e.g., *livers*) have exactly one lemma (e.g. *liver*), exactly one lexeme category (e.g., *plural noun*) and exactly one spelling (e.g., “livers”). Phrase categories (e.g., *noun phrase*) have features (e.g., *number*). Segments (e.g., *noun phrase-head-noun)* have zero or more features and a position. In addition, not all combinations of features and feature values are legal feature-value pairs. They can only be combined in the ways indicated in table 2.

2.2 Language Specific components

The generic linguistic objects and relations defined in section 2.1 are used to define a grammar and a lexicon for a particular language fragment. The definition of the grammar precedes the creation of the lexicon as the grammar specifies first, which features possibly apply to the individual lemma categories and second, which lexeme categories are associated with the lemma categories and what their feature-value pairs are.

2.2.1 Grammar specification

The specification of the grammar of a particular language involves the following: (the second and the third item in this list are optional).

- Specification of the features associated with lemma and phrasal categories, e.g., an English grammar could specify that *Noun* has the feature named *number*, whereas in a Dutch grammar *Noun* would have the features named *gender* and *number*.

- Specification of subcategories of lemma and phrase categories, e.g., the definition of a German grammar requires adding *GenitiveNounPhrase* as a subcategory to the phrase category *NounPhrase*.
- Specification of the default features-value pairs of particular lemma and phrase categories, e.g., a sensible default feature-value pair of *NounPhrase* in many grammars could be: *case=nominative*
- Specification of the lexeme categories that are associated with the individual lemma categories. Each individual lemma category (e.g., *Noun*) has a lexeme category that represents its basic form (*basic:noun*), and possibly additional lexeme categories that represent its inflected forms (e.g., *plural:noun*). For example, in English the basic form of the lemma category *Noun* has the feature-value pair *number=singular*, and in Dutch the basic form of the lemma category *Adjective* has the feature-value pairs *number=singular, definite=-* , and *gender=neuter*.
- Specification of the segments that constitute the rules of syntax of the language, e.g., by defining the segment *NounPhrase-head-Noun,* assigning it to position *5,* and specifying its shared features*: gender, definite,* and *number.*

2.2.2 Populating the Lexicon

After the syntactic features that apply to the individual lemma- and phrase categories have been specified, and the lexeme categories have been defined, the lexicon can be populated. This means that lemmas and lexemes and their features are added to the lexicon. For example, we could add the noun *lever* (liver) to a Dutch lexicon. The syntactic category of this lemma is *noun* and its *gender* is *nonNeuter*. A Dutch noun has two lexeme categories, one to represent its basic form (*number=singular),* and one to represent its plural form (*number=plural*). So the noun *lever* has two lexemes: the basic form (*number=singular*) spelled *lever*, and the plural form (*number=plural*) spelled *levers*. Note that the syntactic category and the feature-value pairs of the lemma are inherited by its lexemes.

2.3 Syntactic tree formation in Segment Grammar

Below I will provide a more detailed account of segments and how they are used within Segment Grammar to create constituent structures of natural language phrases. Within the linguistic community constituent structures of many different sorts are commonly used to describe the syntactic structure of natural language phrases. Both in natural language analysis and in natural language generation constituent structures provide a useful intermediate representation. During the generation process grammar and semantic rules specify how conceptual structures are transformed, first into a constituent structure and subsequently into a string of words that makes up the output phrase. Conversely during parsing the individual words of the input phrase are combined with the grammar and semantic rules to produce a constituent structure, and subsequently to produce a representation of the meaning of the phrase.

Within the linguistic community many different types of grammars are used such as Lexical Functional Grammar, Systemic Grammar, Tree Adjoining Grammar, Montague Grammar, Transformational Generative Grammar, and several Phrase Structure Grammars. Segment Grammar shows some similarities with Lexical Functional Grammar and Tree Adjoining Grammar. It was originally developed as a performance grammar for human sentence production and comprehension. As such it has been used to develop models of the human syntactic tree formation process. There are however several reasons for adopting Segment Grammar in automatic multilingual natural language processing . First it has been applied successfully in the past both in parsers and in generators. Second, it is a relatively simple framework that distinguishes rules of syntactic structure from linear precedence rules. Third, in comparison with other grammars, Segment Grammar produces syntactic structures that show a close resemblance to the conceptual structures as we see them in Grail and other ontological systems.

2.3.1 Syntactic trees

Constituent structure are often presented graphically as a tree (figure 1) . In the following sections I will describe how Segment Grammar is used to produce constituent structures like this. This process is called *syntactic tree formation.*

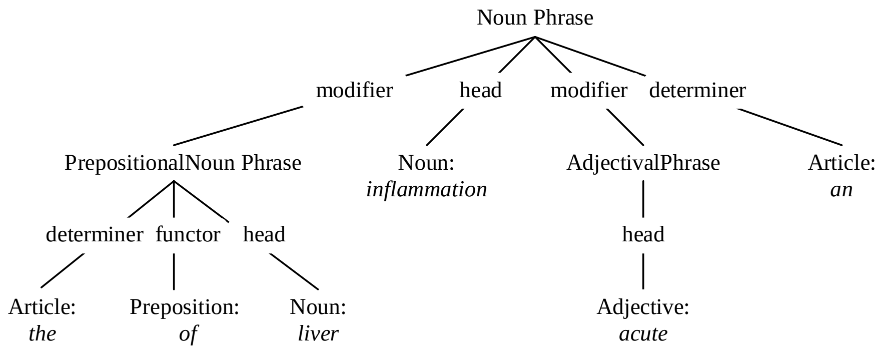


figure 1 Example of a constituent structure of *an acute inflammation of the liver*

2.3.1.1 Segments

Segment Grammar is named after its elementary building blocks which are called segments. A segment is presented visually by a graph consisting of two labeled nodes connected by a labeled arc. Segments are presented in vertical orientation. The top node is called the *root* of the segment and the bottom node is called the *foot* (see figure 2).

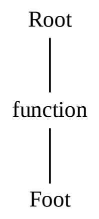


figure 2 Structure of a segment. The top node is called root the bottom node is called foot

The root of a segment is a phrase category, the labeled arc is a syntactic function, and the foot is either a phrase category or a lexical category (see figure 3 below).

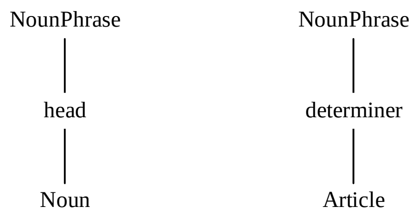


figure 3 Two example segments

Segment Grammar distinguishes five phrase categories[4] (e.g., Sentence, AdjectivalPhrase), fourteen syntactic functions (e.g. head, modifier) and twenty lexical categories (e.g., Noun, Preposition).

Using these categories and functions we could (in principle) create 5 * 14 * (5 + 20) = 1750 different segments. However, the grammar of an individual language will typically have only around fifty different

[4] For purposes that will be explained in more detail in section XX we do not use the commonly applied phrase category *Prepositional Phrase* but introduce a non-standard phrase category we termed *PrepositionalNounPhrase* instead.

segments, and the grammar for a small fragment of a language like the descriptions of surgical procedures will only involve around ten segments.

During the tree syntactic tree formation process, instances of segments are combined into syntactic trees by a process called unification. During this process the root or foot of one segment is merged with the root or foot of another. For example, by unifying the roots of the two segments of (figure 3)we can create a simple tree structure (see figure 4).

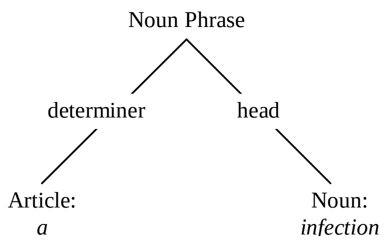


figure 4 Constituent structure after unification of two segments

2.3.1.2   Syntactic features

Next to syntactic and lexical categories and syntactical functions, Segment Grammar distinguishes syntactic features (e.g., *number*, *gender*) and the corresponding feature values (*singular*, *masculine*). Like other grammars Segment Grammar allows only certain combinations of features and values. The syntactic feature *number* for example is allowed to take the values *singular* and *plural* but not the value *genitive*.

The grammar of an individual language specifies which particular features may apply to the syntactic and lexical categories. In English for example nouns have the feature number, but prepositions have not.

Whenever two categories are unified the feature-value pairs of these categories will be unified. For now it suffices to state that during that process the unified category will have the features of both categories. For example if we unify a noun phrase with the feature-value pair *number*=*singular* with a noun phrase with *case*=*genitive* we obtain a noun phrase with the feature-value pairs: *number*=*singular* and *case*=*genitive* (see figure 5). Note that feature-value pairs are presented in a box connected to the constituent they belong to.

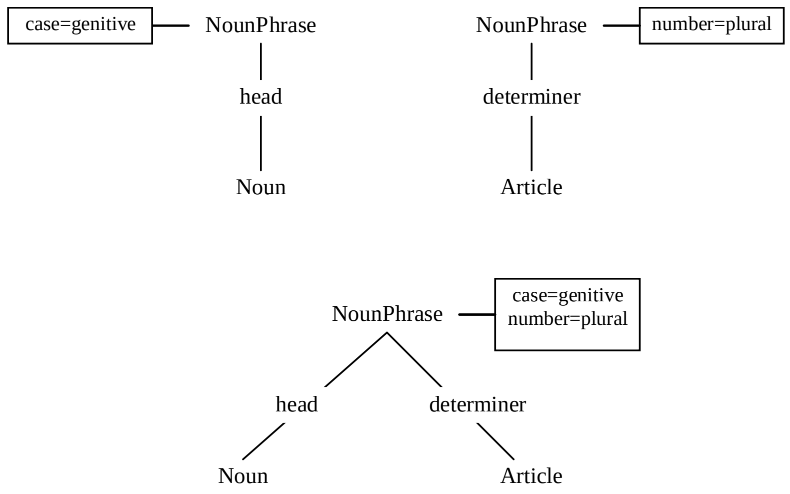


figure 5 Unifying the noun phrases of the segments at the top produces the syntactic tree at the bottom

2.3.1.3 Feature sharing

Clearly the mechanisms described so far are not enough to account for agreement phenomena that exist in many European languages. In English sentences for example, the conjugation of the verb depends (among other things) of value the person of the subject phrase (Viz., *I remove* versus *he removes*). In Segment Grammar agreement is realized by a mechanism called *feature sharing*. In section 2.3.1.1 a segment has been defined by a root, a syntactic function and a foot. To these three elements we add the *set of features* that are *shared* between the root and the foot. For example, a feature that is shared between the root and the foot of the English segment *NounPhrase-head-Noun* is *number*.

When the root and foot share a feature this implies (by definition) that the value of the feature of the foot is equal to the value of the feature of the root, and vice versa. This means that a change to the value of a shared feature (e.g. as the result of unification) will have consequences for both the root and the foot of the segment. For instance, consider the segments unified in figure 6. Assume that the shared feature sets of both segments contain the feature *number*. Before the unification the segment *NounPhrase-head-Noun* has no values for its shared feature *number* (note that the values of shared features are presented in a box that is connected to both the root and the foot of the segment). After unification the value of the feature *number* of *NounPhrase-head-Noun* (and consequently of *Noun*) has been set to *plural*.

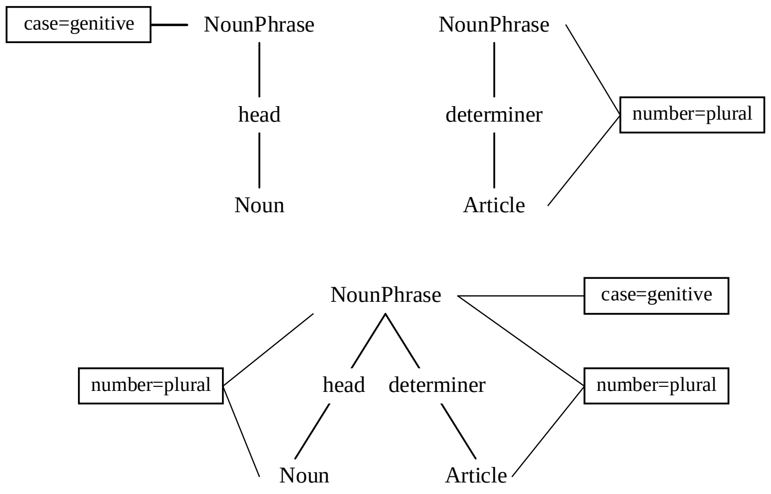


figure 6 Shared features and unification. Unification of the segments at the top produces the tree at the bottom

The syntactic tree formation mechanisms described so far cover the generation of the basic constituent structure of the phrases of a language. Note that before we can actually start constructing constituent structures we have to specify the grammar of that language in terms of the segments and their shared features. In addition we need a lexicon that specifies the words of the language in terms of their syntactic category and features. An example of a constituent structure is presented in (figure 7). Note that the feature values of the individual constituents have been left out for presentation reasons only.

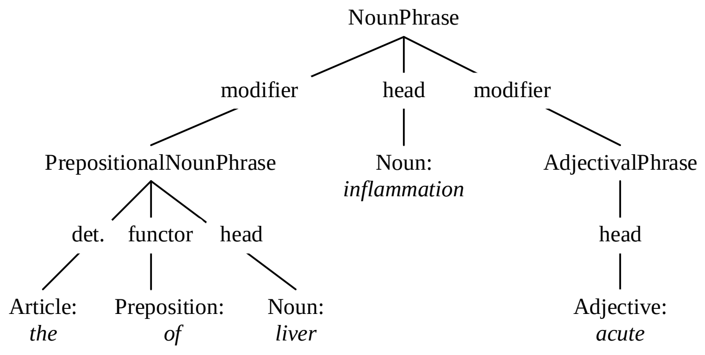


figure 7 Example constituent tree of the phrase: *acute inflammation of the liver*

2.3.2 Word order

Being able to create unordered constituent structures is the first step in producing natural language phrases, assigning order to the constituent structure is the next. For this purpose we now introduce *position* as an element of the definition of a segment. The basic principle is quite simple. The *position* of a segment is a *cardinal number*. This cardinal number indicates the ordinal position of the foot of the segment relative to the other children of the root of that segment. In syntactic tree presentations the position of a segment is presented between parentheses just below the label that represents the function of the segment. Below we present an example of four English segments including their positions.

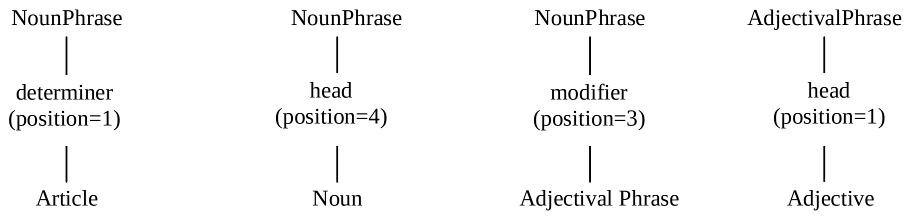


figure 8 Example positions of four English segments

If we apply the ordinal positions of the segment definition to the individual branches of the constituent structure it becomes a constituent tree. The order of the children of each constituent is determined by the definition of the corresponding segment (figure 9).

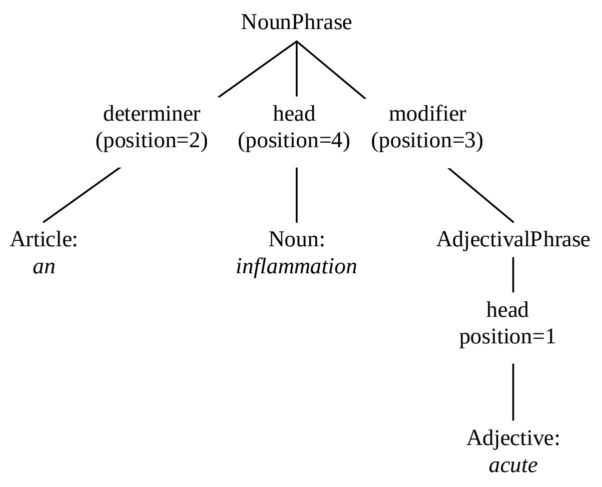


figure 9 Adding order to the constituent structure of the phrase: *an acute inflammation*

Given the ordered constituent tree the sequential ordering of the individual leafs can be derived very easily to produce the phrase *an acute inflammation*.

2.3.3 Word form

Although the example sentences presented so far look quite all right from a grammatical point of view this is mainly due to the fact that they are all in English which is a morphologically simple language. Consider the Dutch phrase * *een acuut ontsteking* which is an ungrammatical translation of the English phrase *an acute inflammation* (figure 9). This phrase is ungrammatical as the uninflected form of an adjective (*acuut*) can only be used in indefinite singular neuter noun phrases. All other noun phrases require the inflected form of the adjective. The constituent tree of *een acute ontsteking* (*an acute inflammation*) is presented in figure 10.

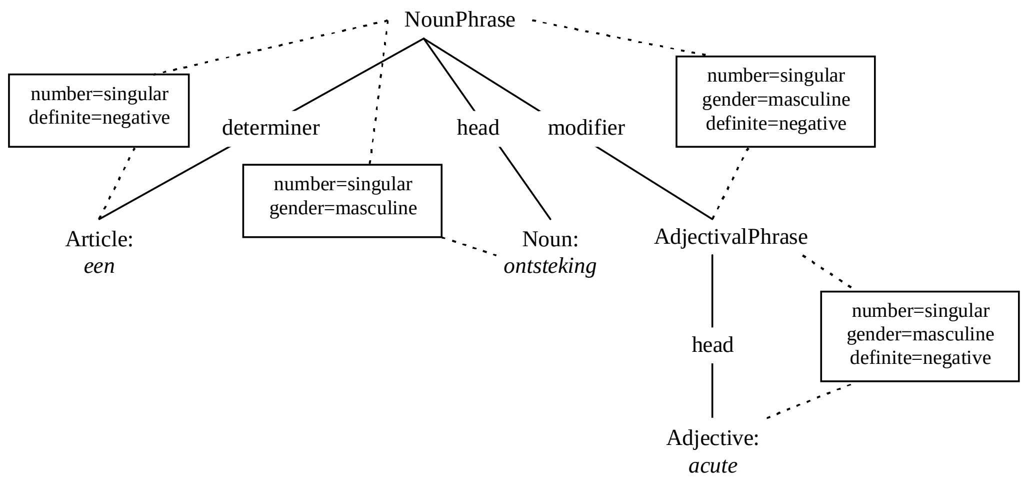


figure 10 The constituent tree of the Dutch phrase: *een acute onsteking* including the feature value pairs of the individual constituents.

In the example we see that the adjective *acute* does not posses the feature-value pairs that justify usage of the uninflected form, hence the inflected form will be selected.

3 Generating phrases to describe Grail concepts

In section 1.3 we explained that producing natural language descriptions of Grail concepts requires first, that the individual concepts of a Grail model are annotated with the individual lemmas of the natural language, and in addition, that the concept composition rules of that Grail model are annotated with the phrase and word combination rules (the Segment Grammar) of the natural language. In section 2 we described a linguistic framework that enables us to represent Segment Grammars and lexicons for multiple languages. In section 3.1 we will first describe the linguistic annotations that represent the semantic relations between Grail concepts on the one hand and linguistic objects (lemmas, segments) on the other. Then, in section **Error! Reference source not found.** we will go into two standard modeling schemes that are commonly applied in Grail, that require the introduction of the notion of filtering. Finally, section 3.3 presents a description of the natural language generation algorithm.

3.1 Annotating grail concepts with linguistic entities

3.1.1 Concept annotations

Producing a natural language phrase that linguistically realizes a Grail concept requires a model of the lexical semantics of the language. This model describes how particular grail concepts can be expressed by a lemma in a particular language. For example the Grail concept *Liver* can be expressed by the Dutch noun *lever.* This type of annotation is called a *concept annotation*. Concept annotations typically involve mappings to lemmas that are traditionally called content words, i.e., nouns, adjectives, adverbs, and verbs.

Note that concept annotations can apply to both primitive and composite Grail concepts. In both cases the lemma mapped to is supposed to describe the concept completely, that is, including all of its defining criteria.

3.1.2 Relation annotations and syntactic frames.

In anticipation of the description of the natural language generation algorithm (in section 3.3) you should know that a composite Grail concept can be described by adding modifier phrases to a natural language phrase that describes one of its formal ancestors. Every defining criterion that distinguishes the concept from this ancestor (often called *topic* in this context*)* will be expressed as a modifier phrase, c.f., (*Fracture which hasLocation Femur)* in section 1.3. How a certain criterion can be expressed using natural language is determined by the so-called *relation annotations*: mappings from *conceptual relations* to *syntactic frames* in the language. Below I will go into relations and syntactic frames respectively.

3.1.2.1 Relations

First, recall that composite Grail concepts are defined by a non-composite concept called *base* and a set of defining criteria (*attribute-value* pairs). A defining criterion can be said to represent a particular *relation* between two Grail concepts. For example the relation called *hasLocation* between the concepts *(Fracture which hasLocation Femur)* and *Femur*. In the following sections relations will be presented as triples of the form *(<topic concept>, <attribute>, <value concept>)*, for example: *(Fracture, hasLocation, Femur).* Note that subsumption relations may exist between relations. For example, the relation *(Fracture, hasLocation, Femur)* is subsumed by the relation: *(Disorder, hasLocation, Bodypart).* Note that relations should be distinguished from Grail statements. In Grail, *sensible* statements are the rules that specify which concepts and attributes can be combined into a composite concept, and *grammatical* statements specify which sensible statements are allowed. Using relation annotations instead of statement annotations to specify how certain defining criteria are to be realised linguistically allows the annotation of relations that have no counterparts as *grammatical* or *sensible* statements in the CORE model.

3.1.2.2 Syntactic frames

In principle, syntactic frames are syntactic constituents of varying complexity. A collection of English example frames is presented in figure 11. A frame has a exactly two phrase categories that will be unified with other constituents during the syntactic tree formation process. They are called the *topic constituent* and the *value constituent.* The are presented in figure 11 in italics and bold face respectively.

In its most simple form a syntactic frame consists of a single segment e.g., the left most frame: *NounPhrase-modifier-AdjectivalPhrase*. Its topic constituent is *NounPhrase* and its value constituent is *AdjectivalPhrase*.

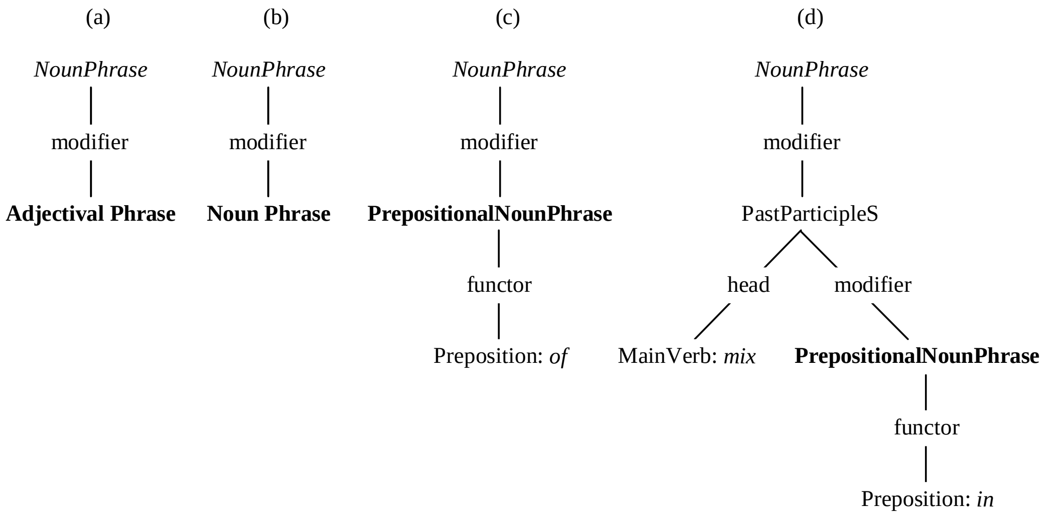


figure 11 Example frames for English. Topic constituents are in italics, value constituents in bold type face

As illustrated in figure 11(c and d), more complex frames may involve additional segments. The role played by syntactic frames is best illustrated by an example. Consider again the concept *(Fracture which hasLocation Femur)*. Suppose that in English the concept *Fracture* is annotated with the noun *fracture*. As a *c*onsequence the concept *Fracture* can be described by the constituent *NounPhrase-head-Noun:fracture*. In addition, suppose that the relation *(Disorder, hasLocation, Bodypart)*, which subsumes the relation *(Fracture, hasLocation, Femur),* is annotated with the frame in figure 11(a): *NounPhrase-modifier-AdjectivalPhrase*. This relation annotation implies that the composite concept's criterion *hasLocation Femur* can be expressed by unifying the constituent describing *Fracture* with the topic constituent of this frame. In addition, an adjectival phrase describing the value of the criterion (*AdjectivalPhrase-head-Adjective:femoral*) can be unified with the value constituent of this frame. This process is illustrated in figure 12. The frame is presented in a square box.

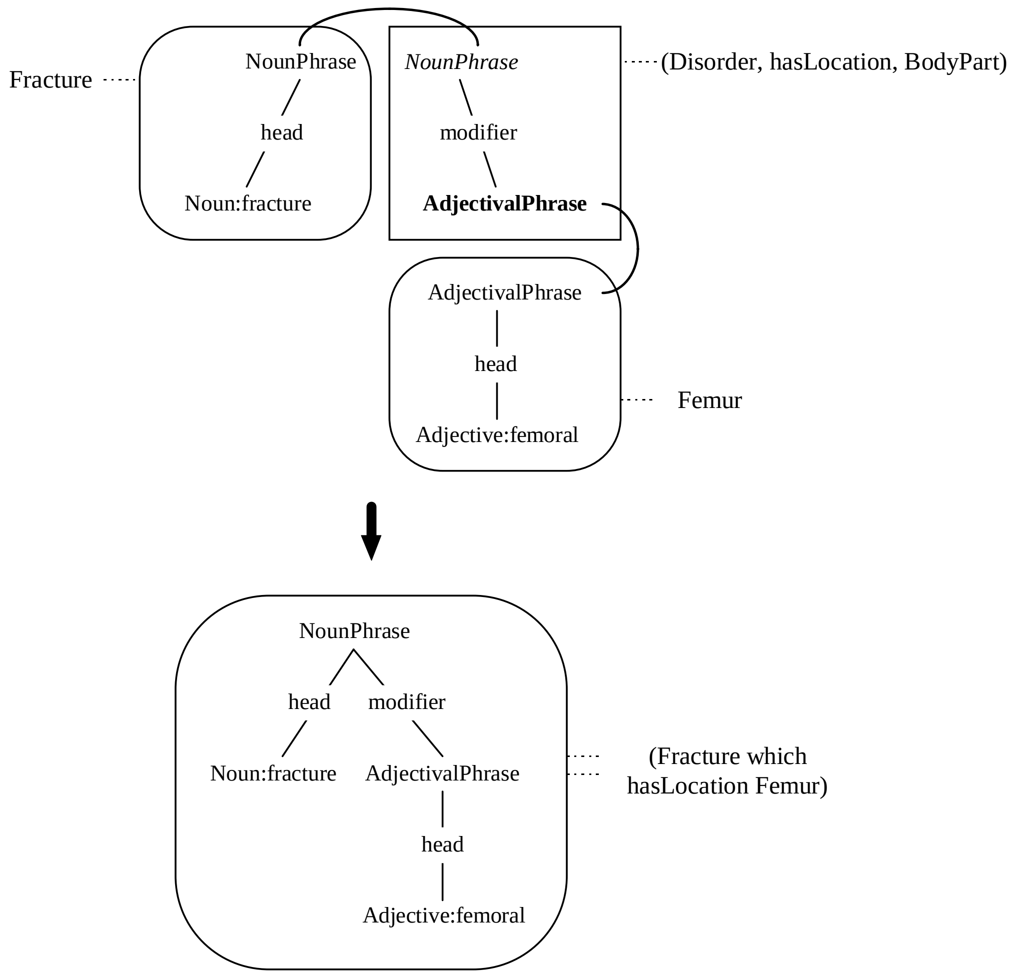


figure 12.Role of a simple syntactic frame in the syntactic tree formation process

Of course, the phrase *femoral fracture* is not the only sensible realization of the concept (*Fracture which hasLocation Femur).* This concept could also be expressed by the phrase *fracture of femur*. This would require an annotation of *(Disorder, hasLocation, Bodypart)* with the frame in figure 11(c). The role of this frame in the syntactic tree formation process is illustrated in figure 13.

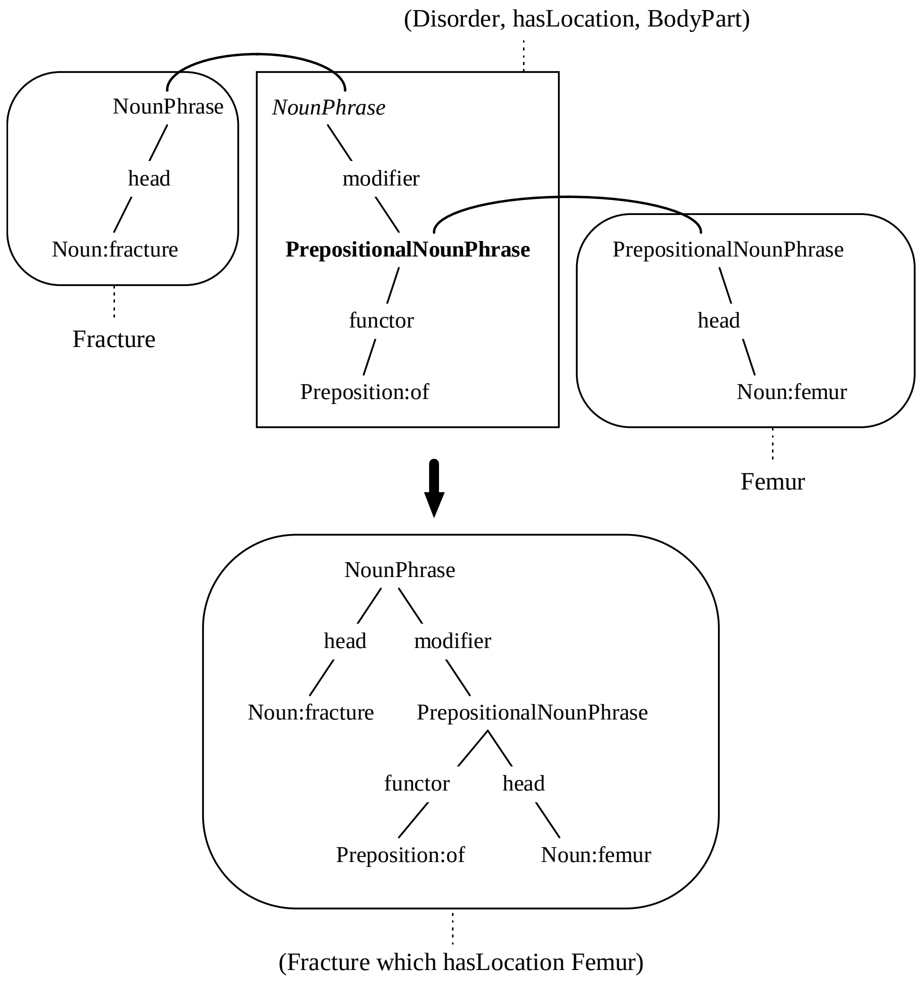


figure 13.Role of a complex syntactic frame in the syntactic tree formation process

3.2 Modeling schemes and filtering

Before I can go into a more detailed description of the natural language generation algorithm a little more needs to be explained about the so-called *modeling schemes* and *filtering*. First, the current CORE model applies composite concepts in a number of standard ways to represent features, processes and surgical procedures to work around certain limitations on the expressiveness of the Grail formalism. Although these and other, similar, representations are essential to the usability of the CORE model, in many cases they would produce natural descriptions that are simply too verbose to be useful. In this section I will describe these structures, why they were introduced, and how we can use the filtering mechanism to produce less verbose natural language for the concepts involved.

3.2.1 Feature-State scheme

In early versions of the CORE model, a concept's features were represented by simple criteriaFor example, *a severe inflammation* used to be represented by:

```
   (Inflammation which <hasSeverity severe>)
```

Although this worked fine in many cases, it became problematic when other things concerning the severity of the inflammation (e.g., the method used to measure it) were to be represented. In order to handle this, features are no longer represented by attributes but by concepts. E.g., the feature *severity* is represented by the concept *Severity* which is a descendant of the concept *Feature*. The value of the feature is represented by particularizing the feature using the attribute *hasState*. A severe inflammation , for example, will be represented by:

```
(Inflammation which <hasSeverity (Severity which < hasState severe >) >)
```

Constructs like this are useful e.g., to represent clinical findings as in:

```
(Inflammation which <hasSeverity (Severity which < hasState severe
                                                   asMeasuredBy methodX >) >)
```

In order to express criteria modeled like this no special arrangements would have to be made if the concepts *(Severity which < hasAbsoluteState mild >)* , *(Severity which < hasAbsoluteState moderate >)* and *(Severity which < hasAbsoluteState severe >)* would be annoated with e.g., the lemmas *Adjective: mild*, *Adjective: moderate* and *Adjective: severe* respectively. However, it would be much more convenient to map these lemmas directly to the individual severity values *mild, moderate,* and *severe* and still produce the same output.

3.2.2 Surgical Procedures

To represent surgical procedures a modeling scheme similar to the feature-state scheme is used. A major modeling problem here was to account for the fact that surgical procedures typically involve multiple (apparently more primitive) surgical deeds such as *removal of abscess involving partial reconstruction of bone tissue*. As Grail does not provide built-in primitives to handle conjunctions, the attributes *isCharacterisedBy* and *isMainlyCharacterisedBy* have been introduced to produce concepts such as presented in figure 14.

```
(Process which <
    hasClinicalRole SurgicalRole
   isMainlyCharacterisedBy (Removing which <
                              hasClinicalRole SurgicalRole
                              actsSpecificallyOn Abscess >)
   isCharacterisedBy (Reconstructing which <
                         hasClinicalRole SurgicalRole
                          actsOn Bone >) >)
```

figure 14 Modeling scheme to represent surgical procedures

The second problem with the representation of surgical procedures was that Grail has no built-in mechanism to handle negations, so surgical procedures that involve the exclusion of a particular deed could not be represented easily. For this purpose another scheme has been introduced. This layer wraps up an individual *SurgicalDeed* in a composite concept using the concepts *Performance* or *NonPerformance* and the attribute *isEnactmentOf* to produce concepts like the one shown in figure 15:

```
(Process which <
   hasClinicalRole SurgicalRole
   isMainlyCharacterisedBy
     (performance which <
        isEnactmentOf
          (Removing which <
             hasClinicalRole SurgicalRole
             actsSpecificallyOn
               (Abscess which <
                  hasLocation ExternalEar
                  hasSeverity (Severity which <
                                 hasAbsoluteState severe >) >)
             hasExtend complete >) >)
   isCharacterisedBy
     (nonPerformance which <
        isEnactmentOf (Incising which <
                         hasClinicalRole SurgicalRole
                         actsSpecificallyOn Pinna >) >) >)
```

figure 15 The use of wrapper concepts in the conceptual representation of a surgical procedure

Although these modeling schemes are very useful as a representation mechanism, they complicate the natural language generation proces considerably. For example, a natural language phrase that describes the surgical procedure presented in figure 15 in a straightforward manner is *surgical procedure that involves performing complete surgical removal of a severe abscess of the external ear and that involves not performing surgical incising of the auricle.* Although this phrase accurately describes the concept, in most contexts a less verbose description such as *complete removal of a severe abscess of the* *external ear without incising of the auricle* will be preferred. However, in order to produce such less verbose phrases we would have to add *concept-lemma* mappings for all concepts of the form *(Process which < isMainlyCharacterisedBy (performance which < isEnactmentOf X >) >)* which would be very inconvenient and time consuming. Instead we would prefer to add such mappings only to the descendants of *SurgicalDeed* that could be substituted for *X*.

3.2.3 Unwanted criteria

Next to the modeling schemes des cribed above, the CORE model features certain defining criteria that would produce unwanted results when expressed by the natural language generator. These criteria involve e.g., tha attributes *ApplicationAttribute*, *RoleDesignatingAttribute* and their descendants. Although they distinguish a concept from its ancestors they are not suppose to have any effect on the way the concept is described using natural language. Consider for example, the concept *(Closing which hasClinicalRole SurgicalRole)*. It could be described very well by the phrase *surgical closing*, but in many contexts the adjective *surgical* would be redundant and the phrase *closing* is preferred. In principle this could be handled by annotating both concepts with the noun *closing*. However, if we prefer never to express the difference between general actions such as opening, closing, drilling, shaving etc. and their surgical counterparts we would have to add concept annotations for each of them. For this reason we introduce the notion of *suppressing* which will be explained in section 3.2.4.2.

3.2.4 Tagging and filtering

To provide a solution to the problems concerning modeling schemes and unwanted criteria described in the previous sections we use the notions of *wrapper* and *suppression tagging* and *filtering*.

3.2.4.1 Wrapper tagging

The natural language generation problems with both feature-state and surgical procedure concepts are solved by a single mechanism. First, the relations involved in these modeling schemes are tagged as so-called *wrappers*. Second, the natural language generator applies a filter mechanism to *unwrap* concepts that involve such a wrapper relation. Tagging a certain relation *(Topic, attribute, Value)* as a wrapper implies tagging all the relations that involve descendants of *Topic*, *attribute*, and *Value* as a wrapper. For example, assume that the relation *(Feature, hasState, State)* has been tagged as a wrapper. The filter mechanism will unwrap the concept *(Severity which < hasAbsoluteState severe >)* to produce the concept *severe*, as *Severity* is a descendant of *Feature*, *hasAbsoluteState* is a descendant of *hasState*, and *severe* is a descendant of *State*.
The filter mechanism is also applied to the surgical procedures described in 3.2.2. Recall that surgical procedures involve two relations: *(Process, isMainlyCharacterisedBy, performance)* and *(Enactment, isEnactmentOf, SurgicalDeed)*. To unwrap a surgical procedure requires tagging both relations as wrappers. During the natural language generation process the filter mechanism wil unwrap the concept *(Process which < isMainlyCharacterisedBy (performance which < isEnactmentOf Removal >)* in two steps. First it unwraps the concept to *(performance which < isEnactmentOf Removal >)*. Subsequently, *(performance which < isEnactmentOf Removal >)* is unwrapped to produce the concept *Removal*. (*performance* is a descendant of *Enactment* and *Removal* is a descendant of *SurgicalDeed*).

3.2.4.2 Suppression tagging

In order to prevent the linguistic realisation of certain criteria for certain concepts, relations can be tagged to be *suppressed*. For example, in order to prevent the expression of the criterion *hasClinicalRole-SurgicalRole* in the example presented in 3.2.3, the relation *(SurgicalDeed, hasClinicalRole, SurgicalRole)* is tagged as a relation that is to be suppressed. The filter mechanism will simply hide the criterion *hasClinicalRole-SurgicalRole* from the list of defining criteria of *(Closing which hasClinicalRole SurgicalRole)*.

3.2.4.3 Example

To illustrate the filter process figure 16 shows step by step how the concept at the top is filtered to produce the concept at the bottom.

```
(Process which <
   playsClinicalRole SurgicalRole
   isMainlyCharacterisedBy
     (performance which <
        isEnactmentOf
          (Removing which <
             playsClinicalRole SurgicalRole
             actsSpecificallyOn
                (Abscess which <
                   hasLocation ExternalEar>
                   hasSeverity (Severity which <
                                  hasAbsoluteState severe >) >)
             hasExtend complete >) >)
   isCharacterisedBy
      (nonPerformance which <
         isEnactmentOf (Incising which <
                          playsClinicalRole SurgicalRole
                          actsSpecificallyOn Pinna >) >)

(Process which <
   isMainlyCharacterisedBy
     (performance which <
        isEnactmentOf
          (Removing which <
             actsSpecificallyOn
                (Abscess which <
                   hasLocation ExternalEar>
                   hasSeverity (Severity which <
                                  hasAbsoluteState severe >) >)
             hasExtend complete >) >)
   isCharacterisedBy
      (nonPerformance which <
         isEnactmentOf (Incising which < actsSpecificallyOn Pinna >) >)

(performance which <
   isEnactmentOf
      (Removing which <
         actsSpecificallyOn
            (Abscess which <
               hasLocation ExternalEar
               hasSeverity (Severity which <
                              hasAbsoluteState severe >) >)
         hasExtend complete >)
     isCharacterisedBy
        (nonPerformance which <
           isEnactmentOf (Incising which < actsSpecificallyOn Pinna >) >) >)

(Removing which <
   actsSpecificallyOn
      (Abscess which <
         hasLocation ExternalEar
         hasSeverity (Severity which < hasAbsoluteState severe >) >)
   hasExtend complete
   isCharacterisedBy
      (nonPerformance which <
         isEnactmentOf (Incising which < actsSpecificallyOn Pinna >) >) >)

(Removing which <
   actsSpecificallyOn
      (Abscess which <
         hasLocation ExternalEar
         hasSeverity severe >)
   hasExtend complete
   isCharacterisedBy
      (nonPerformance which <
         isEnactmentOf (Incising which < actsSpecificallyOn Pinna >) >) >)
```

figure 16 Unwrapping wrapped concepts

3.3 The Natural Language Generation algorithm

The input to the generation algorithm consists of a Grail concept, a target language, a parameter that specifies whether or not articles should be used, and a parameter that specifies the intended phrase category. The output is a phrase that expresses the input concept in the target language, and some error diagnostics .

The concepts in a Grail model represent classes of objects rather than individual instances. For example, the concept *(Fracture which < hasLocation Femur >)* does not refer to a particular femur fracture occurring in a particular patient. Instead it refers to any fracture of any femur. Grail concepts correspond with *types* rather than with *tokens*. As a consequence the natural language generation algorithm produces phrases that have generic reference. Although most European languages have multiple ways of expressing generic reference the present generator applies the singular indefinite form. References to named parts of the body are realized by the singular definite form.
The output phrase is produced in three steps (see figure 17). First, a constituent structure is generated for the input concept. Then the constituent structure is serialized to produce a sequence of lemmas. Finally the spellings of the appropriate word forms of the lemmas are concatenated to form the surface string. These steps are described in the sections 3.3.1 to 3.3.3 .

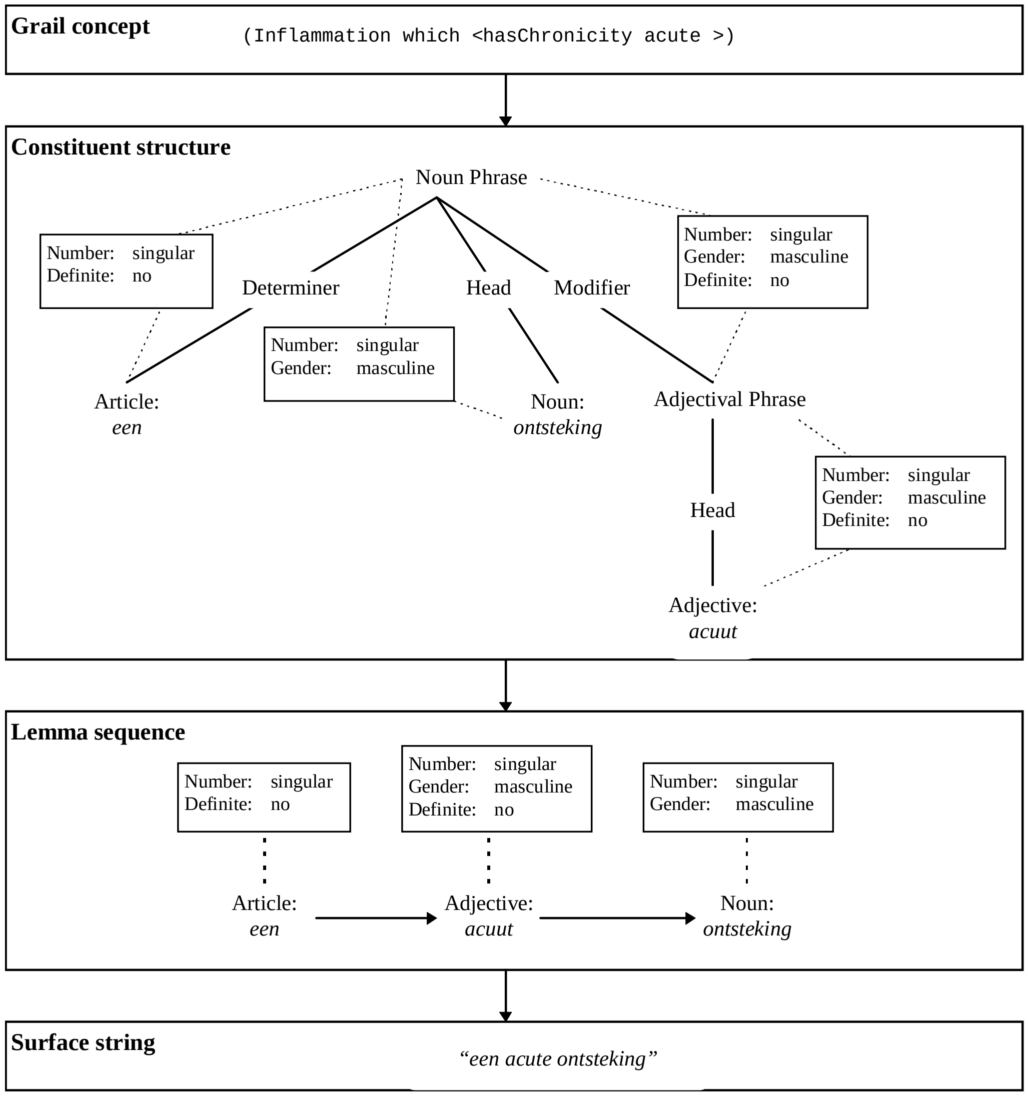


figure 17 From Grail concept to natural language phrase

3.3.1 Generating the constituent structure

The algorithm used to produce a constituent structure to describe a Grail concept *Concept* using a phrase of type *Phrase category* in a language *Language* is described below in pseudo code. The basic algorithm consists of two procedures that call each other recursively. The first is called *GenerateConstituent*, the other *ExpressCriterion*. Variable names are presented in italics.

```
PROC GenerateConstituent (Concept, PhraseCategory, Language, AddArticles) : Constituent
1) LemmaCategory :=
         SyntacticCategory(foot(GetSegment(Language, PhraseCategory, head, *)));
   Lemma := GetAnnotation(Concept, Language, LemmaCategory);
   IF    Lemma EQUALS NIL
   THEN  Base := filter(base(Concept))
         Criteria := filter(criteria(Concept))
         Look for concept annotations (with category LemmaCategory) at ancestors of
         Concept with a base identical to Base.
         IF    annotations exist
         THEN  Ancestor := most specific ancestor with annotation;
               Lemma := GetAnnotation(Concept, Language, Ancestor);
               Criteria := RemoveCriteria(Criteria, GetCriteria(Ancestor);
         ELSE  Look for annotations at ancestors of Concept that have a
               base other than Base
               IF    annotations exist
               THEN  Lemma := the lemma that maps to the most specific ancestor
                     Criteria := NIL
               ENDIF
         ENDIF
   ENDIF
2) SegmentInstance := CreateSegmentInstance(PhraseCategory, head, Lemma)
3) Constituent := root(SegmentInstance)
4) FOREACH  tuple <Attribute, Value>  IN Criteria  DO
      ExpressCriterion(Language, Concept, Constituent, AddArticles, Attribute, Value)
   ENDFOR
5) IF    AddArticles = TRUE AND
         GetSegment(Language, NounPhrase, determiner, Article) NOT EQUALS NIL AND
         Constituent has no branch of the form [Constituent, determiner, *] AND
         Constituent has no feature determinable: -
   THEN  DeterminerConstituent :=
            CreateSegmentInstance (NounPhrase-determiner-Article);
         Constituent := Unify(Constituent, root(DeterminerConstituent);
   ENDIF
6) RETURN Constituent
END PROC

PROC ExpressCriterion(Language, Concept, Constituent, AddArticles, Attribute, Value)
1) Frame := retrieve the most specific frame for the relation (Concept, Attribute,
            Value) in Language, such that the SyntacticCategory(TopicConstituent(Frame))
            is compatible with the SyntacticCategory(Constituent).
2) FrameInstance := Instantiate(Frame);
2) ValueConstituentCategory := SyntacticCategory(ValueConstituent(FrameInstance));
3) ValueConstituent :=
      GenerateConstituent (Value, ValueConstituentCategory, Language, AddArticles);
4) IF    ValueConstituent NOT EQUALS NIL
   THEN  Unify(Constituent, TopicConstituent(FrameInstance));
         Unify(ValueConstituent(FrameInstance), ValueConstituent);
   ELSE  Repeat from step 1 with other frame;
   ENDIF
END PROC
```

3.3.2 Serializing the constituent structure

The input to this process is an unordered constituent structure. The constituent structure is processed starting from the root. The root and its descendants are recursively expanded in the order that is specified by the *positions* of the segments that were used to create the branches. Note that there is a special way to handle segments that involve the same basic syntactic categories and functions but differ with respect to their serial position. In French for example, the default serial position of the segment *NP-modifier-ADJP* is after the segment *NP-head-Noun*. However, some French adjectives are prenominal instead postnominal modifiers. In order to handle this the generic language model allows the definition of segments based on other segments, e.g., the segment *NP-modifier-PrenominalADJP* as a child to *NP-modifier-ADJP*. This segment can be defined to have a serial position which comes before *NP-head-Noun*. Note that *PrenominalADJP* is an subcatategory of *ADJP* with the feature-value pair *prenominal*=+. Now every adjective in the lexicon that has this feature as well will automatically turn up before the noun. Of course the grammar must enable the feature *prenominal* for the categories *Adjective* and *ADJP*, and *prenominal* must be a shared feature of the segment *ADJP-head-Adjective* in order to enable the *prenominal* feature of a particular adjective to pass up to its parent constituent *ADJP*.

3.3.3 Producing the surface string

The input to this process is a sequence of lemmas. First, every lemma in the sequence is substituted by the spelling of one of its lexemes that has the right syntactic features. The individual spellings are concatenated, inserting spaces as required. No spaces will be inserted between a word and its prefixes or suffixes. A word can become a prefix or suffix through the feature *affixRole* which takes one of the values *prefix*, *infix*, or *suffix*. In the grammar this requires first, to enable the *affixRole* feature for the lexical and phrasal categories involved (e.g., *Noun* and *NP*), second, the definition of a phrase category with that feature e.g., *PrefixNP*, third, the definition of a segment which triggers the affix (e.g., *NP-modifier-PrefixNP*), and fourth, adding *affixRole* to the set of shared features of the appropriate lexical segment (*NP-head-Noun*) in order to pass it down to the lexical level.

Finally the surface string should be processed in a language dependent way that accounts for phenomena such as the usage of *a* versus *an* in English and the contraction of *le* into *l'* in French depending on the pronunciation of the word that follows, and e.g., the contraction of two words such as *de le* into *du* in French and *in het* into *er* in Dutch. As the current version of the generation software has no built-in provisions for substitutions like this, they should be implemented by the client application itself.

4 References

[Kempen 87]	Kempen, G. (1987) A framework for incremental syntactic tree formation. In: Proceedings of the Tenth International Joint Conference on Artificial Intelligence (IJCAI'87), Milan.
[Kempen ea 87]	Kempen, G., and Hoenkamp, E. (1987). An incremental procedural grammar for sentence formulation. Cognitive Science, 11, 201-258.

Generating Multilingual Natural Language Expressions for Grail Concepts

Part II: Implementation

Wim Claassen
University of Nijmegen
NICI

Contents

1 Introduction.. 1
2 Using the ROIS system development kit 1
    2.1 Object types 1
    2.2 Network Programming Language... 1
    2.3 NPL modules used in the current implementation.. 2
3 Language models.. 2
    3.1 Representation of basic syntactic objects 2
    3.2 Representation of Grammars 4
    3.3 Representation of lexicons.. 6
    3.4 Representation of Syntactic trees 8
    3.5 Representation of syntactic frames.. 8
    3.5.1.1.1.1.1.11 Overview of the implementation of the generic linguistic framework... 9
4 Representation of relations in Grail 10
5 Linguistic annotation of Grail models 10
6 Filtering: criterion suppression and wrappers... 11
7 Implementation of the generation algorithm.. 11
    7.1 Generate 12
    7.2 Syntax.. 12
    7.3 Semantics 12
8 API of Natural Language Generation modules 12
    8.1 Features.. 12
    8.2 Lexicon.. 12
    8.3 Segments 12
    8.4 Constituents... 12
    8.5 Syntax.. 13
    8.6 Frames... 13
    8.7 GrailExtension 13
    8.8 Semantics 13
    8.9 Generate 13
9 External/interchange formats for Language models and Semantic models 13

1 Introduction

The documentation on the ROIS based Natural Language Generator developed within the context of the Galen In Use project consist of three papers: (1) a theoretical paper on the design of the generator, (2) a description of its implementation, and (3) a manual describing how the generator is to be used.
This paper descibes the ROIS implementation of the generator.

2 Using the ROIS system development kit

This section provides an introductory overview of ROIS. For a thorough description of ROIS please consult the document ROIS: a knowledge server (van der Haring, 1996). ROIS is a server application that provides support to create and manipulate complex data structures that can be represented as graphs. The ROIS server is programmed using *Network Programming Language* (NPL). The ROIS development tool kit consists of the *ROIS server*, the *ROIS debugger*, The ROIS Network Programming Language Compiler *Idefix* , and *Mole* which is a ROIS client application that provides a low-level view on ROIS graphs, and an interface to call ROIS client tasks. ROIS graphs are stored in data files that are called *models*.

2.1 Object types

ROIS graphs consist of objects of two basic types called *nodes* and *links*. There are three types of nodes: *node classes*, *link classes* and *qualifiers*. ROIS *links* consist of a tail node, a link class, a qualifier, and a head node, often presented as a quadruple (tail node, link class, qualifier, head node).
Among many other things, ROIS provides mechanisms for inheritance, basic inferencing, and type and cardinality restrictions on link classes. It also supports the creation of *instances* of node classes.

2.2 Network Programming Language

The ROIS Network Programming Language (NPL) is a language that has a syntax similar to Modula 2. It provides constructs to create and modify objects of the types described in the previous section. In addition, it provides constructs to search, test and select graphs. ROIS allows users to define their own procedures which are called *tasks*. ROIS tasks are defined within named modules. Tasks may call each other recursively and they can be marked as *hidden* or *client* tasks. The former are only accessible to tasks within the same module whereas the latter can be accessed (called) by ROIS client applications. Unmarked tasks are available to all other modules, but not to ROIS client applications.
The ROIS NPL compiler *Idefix* compiles NPL modules to produce ROIS Virtual Machine (RVM) code. For a description of Idefix please refer to the Idefix Reference manual. A module's RVM code is loaded dynamically by the ROIS server when a ROIS client application calls a task from that module.
Building a ROIS client application typically involves the following stages:
First the structure of the graphs used to implement the objects that are relevant in the client application(s) are defined using Idefix. This is usually done by defining a task *createModel* that adds the high-level node and link classes, and the qualifiers to a module's basic model. Subsequently a number of creator, selector and destructor tasks will be defined. After compilation the tasks defined can be tested by calling them from the ROIS client application Mole. If required, the debugger presents debug information to the NPL programmer. After the modules have been defined and tested a client application is built to call the *client* tasks defined by the NPL programmer. Documentation on the ROIS client API is available on the Web. The data flow of this process is illustrated in figure 18.
The natural language generation module uses a ROIS implementation of Grail that is available to clients as *Grail.RVM*. It consists of a collection of client tasks that implement the Grail formalism. The client applications that have been developed to create and maintain Grail models are called *GCE* (*Galen Case Environment*) and *GCE Workspace*. A description of these tools can be found on the Web.

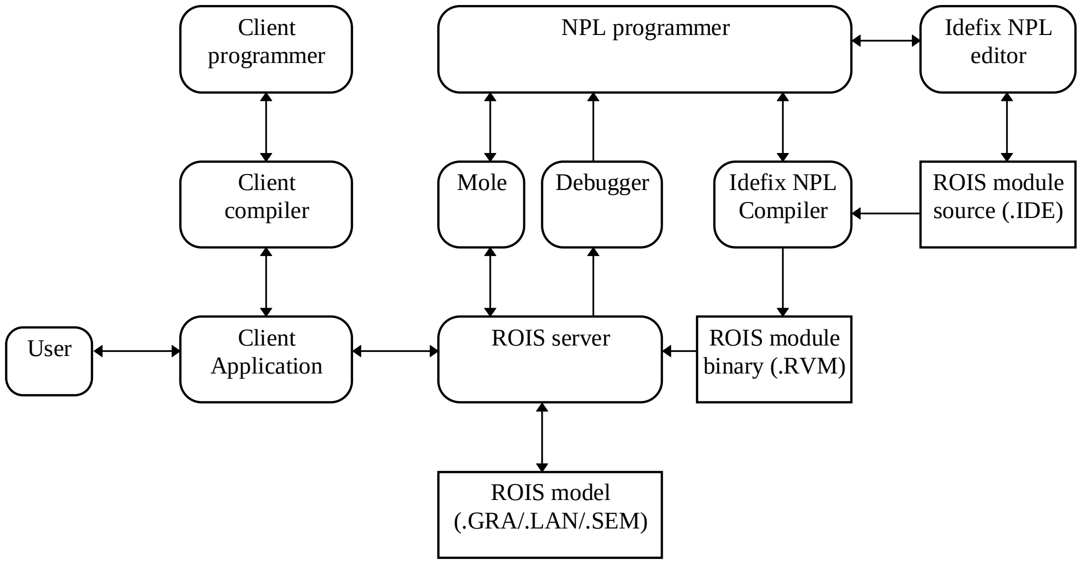


figure 18: Data flow diagram of ROIS system development process.

2.3 NPL modules used in the current implementation

The modules that implement the natural language generation component are described in table 3.

Module name	**Description**
Generic	definition of basic language model
Features	potential and actual features
Lexicon	lemma categories; lemmas; lexeme categories; lexemes; spellings
Segments	representation of segments
Constituents	constituent trees
Syntax	processing constituent trees; unification
Frames	representation of syntactic frames
Grail	retrieval of defining criteria of concepts (including Filtering)
GrailExtension	tagging wrapper concepts; testing and manipulating criteria sets
Semantics	basic semantic model; mappings between grail and language models (annotations)
Generate	main tasks

table 3: Modules that implement the natural language generation component

3 Language models

Both the generic and the language specific linguistic information that is required for multilingual natural language processing in ROIS are represented as data in ROIS models called *language models* (data files with extension .LAN). There will exist one language model for each language covered. These models consist of a built-in generic part that is identical in all models, and a language specific part for each language that is build by defining the actual grammar, adding words, and word forms using Humpty.

3.1 Representation of basic syntactic objects

The generic linguistic framework of elementary linguistic objects and elementary linguistic relations has been implemented by creating a set of qualifiers, a hierarchy of ROIS node and link classes and by providing several ROIS tasks that can be used to create the grammar and a lexicon for a particular language. The top part of the node class hierarchy is shown in figure 19.

```
topNode
  topNodeClass
    SyntacticObject
      Constituent
        LemmaCategory
          Noun
            basic:Noun
            plural:Noun
          Article
            basic:Article
             singular indefinite:Article
            ..
          Adjective
            basic:Adjective
            uninflected:Adjective
          Preposition
            basic:Preposition
           ..
        PhraseCategory
          NounPhrase
          AdjectivalPhrase
          PrepositionalNounPhrase
          ..
      FeatureValue
        GenderValue
          Masculine
          Feminine
          Neuter
        ..
```

figure 19: Top part of the node class hierarchy

Syntactic objects come in four kinds: lemma categories, lexeme categories, phrase categories and feature values. The subclasses of *LemmaCategory* (*Noun*, *Article etc.)* and *PhraseCatgeory* (*NounPhrase*, *Adjectival Phrase*, etc.) correspond with the word and phrase categories that are traditionally used within the linguistic community. The same holds for the node classes that represent feature values (*Masculine*, *Feminine*, etc.). Lexeme categories are children of their corresponding Lemma category, e.g., *basic:Noun* and *plural:Noun*, and correspond with the different word forms that may exist for each lemma category. The top part of the link class hierarchy is shown in figure 20:

```
  topLinkClass
    function
      head
      modifier
      determiner
      functor
      ..
    feature
      binaryFeature
        definite
      gender
      person
      number
      case
      ..
    spelling
```

figure 20: Top part of the link class hierarchy

We distinguish three general kinds of syntactic relations: *function* (e.g., *head*, *modifier*, etc.), *feature* (*gender*, *person*, etc.) to represent syntactic functions and syntactic features respectively, and *spelling* to associate lexemes with their spelling.
The generic linguistic framework contains high level sanctions involving the node and link classes described above. They are created using a sanctioning qualifier called *general*. The following NPL code (figure 21) creates that qualifier and adds the links to represent on a general level that phrase categories can have two kinds of constituents: phrase categories and lemma categories.

```
UPDATE @Model
    ADD QUALIFIER 'general'
        SANCTIONED BY systemSanction
        PROPERTIES sanctioning
        INHERITANCE normal
    ADD LINK PhraseCategory.hasConstituent.general.PhraseCategory
    ADD LINK PhraseCategory.hasConstituent.general.LemmaCategory
```

figure 21: NPL code to create *general* qualifier

There is no single *general* sanctioning relation between *Constituent* and *FeatureValue*. Instead for every subclass of *feature* a *general* sanctioning link is created between *Constituent* and the corresponding *FeatureValue*. For example to represent that the gender of a constituent can be masculine feminine or neuter (figure 22).

```
UPDATE @Model
  ADD LINK Constituent.gender.general.GenderValue
  ..
```

figure 22: Sanctioning of feature values

These sanctioning links of the form *Constituent.feature.general.FeatureValue* (one for each subclass of *feature*) are used in the specification of the language specific sanctions as will be described in section 3.2 below.

3.2 Representation of Grammars

The task *Syntax.createModel* creates a generic language model that incorporates the linguistic framework described above including four additional qualifiers that support the specification of grammars for particular languages, and the syntactic tree formation process described in Part I of the documentation. The specification of the grammar of a particular language fragment involves (among other things) the specification of the syntactic features that may apply to particular lemma and phrasal categories, and the specification of the segments that are required to cover a relevant fragment of the language. In order to support the specification of the grammar of an individual language the generic language model incorporates the sanctioning qualifiers called *potential* and *segment.* Both qualifiers are sanctioned by the qualifier *general* (figure 23).

```
UPDATE @Model
    ADD QUALIFIER 'potential'
        SANCTIONED BY general
        PROPERTIES sanctioning
        INHERITANCE normal
    ADD QUALIFIER 'segment'
        SANCTIONED BY general
        PROPERTIES sanctioning
        INHERITANCE normal
```

figure 23 Fragment of NPL code that creates the qualifiers *potential* and *segment*.

The qualifier *potential* is used to associate features with the lemma and phrase categories to which they apply in a certain language. For this purpose the NPL task *Features.AddPotentialFeatures* has been defined (figure 24).

```
CLIENT TASK AddPotentialFeatures @Model %cat `features;
  DEFINE %feature;
  DEFINE `values;
  DEFINE %value;
BEGIN
  FOREACH NODECLASS %feature IN `features DO
    `values := NIL ;
    // Get the potential value of the feature
    FROM @Model TO `values
      SELECT NODES Constituent.%feature.general.?value;
    FOREACH NODECLASS %value IN `values DO
      UPDATE @Model
        ADD LINK %cat.%feature.potential.%value;
    ENDFOR;
  ENDFOR;
END AddPotentialFeatures;
```

figure 24 Definition of the task *Features.AddPotentialFeatures*

This task associates a number of features with the input category (*%cat*). For each feature it first looks up the potential value as represented in the generic model, and subsequently creates a link between the input category and potential value. For example, in order to specify that nouns can have the feature number the task should be called with parameters: %cat = noun and `features = number. As a result the link: *noun.number.potential.NumberValue* would be added to the model.
The qualifier *segment* is used to represent segments in a way that is more complex than one would expect. If a segment would be represented by a link of the form *root.function.segment.foot* we would be unable to represent its shared features and it destination, as ROIS currently does not support links between links. On the other hand, links of the form *root.function.segment.foot* are useful for a number of reasons. As they are inherited by all the descendants of the root and the foot retrieval of the segments of a language via the root or the foot becomes an easy task. In addition links of this form are needed anyway as they have to sanction the creation of links that represent the branches of syntactic trees.
In order to work around this problem the current implementation applies a 'dual' representation of segments. This is illustrated in the definition of the task *Segments.AddSegment* (figure 25).

```
CLIENT TASK AddSegment @Model %root %function %foot `sharedFeatures;
  DEFINE %segment;
  DEFINE $name;
  DEFINE %sharedFeature;
BEGIN
  // Add segment involves four actions:
  // 1 make link class below %function. The name
  //   of this link class is %root-%function-%foot e.g., 'NP-head-Noun'
  // 2 add sanction: %root.%segment.segment.%foot
  // 3
  // 4 add shared features to segment
  MakeSegmentName %root %function %foot $name;
  UPDATE @Model
    // Add the segment as a LinkClass
    ADD LINKCLASS $name %segment
      PARENTS %function Segment

      CARDINALITY MANY ONE
    ADD LINK %segment.hasRoot.internal.%root;
    ADD LINK %segment.hasFoot.internal.%foot;
    // Add  a link to sanction %root.%segment.constituent.%foot;
    // needed to build constituent trees
    ADD LINK %root.%segment.segment.%foot
    // Add shared features
  ;
  FOREACH NODECLASS %sharedFeature IN `sharedFeatures DO
    UPDATE @Model
      ADD LINK %segment.hasSharedFeature.segmentProperty.%sharedFeature;
  ENDFOR;
END AddSegment;
```

figure 25: Definition of the task *Segments.AddSegment*

First, this task creates the segment as a link class with the name "<root>-<function>-<foot>" as a subclass to <function>. Second, it creates the link <root>.<segment>.segment.<foot>. This link sanctions the creation of links of class <segment> with qualifier *constituent* between descendants of <root> and descendants of <foot>. Third, two links with qualifier *internal* are created to make retrieval of the segment's root and foot more efficient. And finally, the segment's position and shared features are represented by creating links between the segment and the features, using the link classes *hasPrimaryDestination* and *hasSharedFeature* and qualifier *segmentProperty*. These link classes and this qualifier are actually represented in the generic language model, but they exist for implementation reasons only.

```
CLIENT TASK AddSegment @Model %root %function %foot `sharedFeatures;
  DEFINE %segment;
  DEFINE $name;
  DEFINE %sharedFeature;
BEGIN
  // Add segment involves four actions:
  // 1 make link class below %function. The name
  //   of this link class is %root-%function-%foot e.g., 'NP-head-Noun'
  // 2 add sanction: %root.%segment.segment.%foot
  // 3
  // 4 add shared features to segment
  MakeSegmentName %root %function %foot $name;
  UPDATE @Model
    // Add the segment as a LinkClass
    ADD LINKCLASS $name %segment
      PARENTS %function Segment

      CARDINALITY MANY ONE
    ADD LINK %segment.hasRoot.internal.%root;
    ADD LINK %segment.hasFoot.internal.%foot;
    // Add  a link to sanction %root.%segment.constituent.%foot;
    // needed to build constituent trees
    ADD LINK %root.%segment.segment.%foot
    // Add shared features
  ;
  FOREACH NODECLASS %sharedFeature IN `sharedFeatures DO
    UPDATE @Model
      ADD LINK %segment.hasSharedFeature.segmentProperty.%sharedFeature;
  ENDFOR;
END AddSegment;
```

figure 26 Definition of the task *Segments.AddSegment*

3.3 Representation of lexicons

The structure of the lexicons used by the natural language generation module is identical for all languages. However, the grammar specifies which features can apply to the lemmas of particular categories. Lemmas are represented as subclasses to the particular lemma category they belong to. Their (internal) name is constructed by concatenating the spelling of their basic form and their category. Adding a lemma also involves adding its basic form. This is illustrated by the definition of the task *Lexicon.AddLemma* (figure 27).

```
CLIENT TASK AddLemma @Model $basicSpelling %category :%NewLemma ;
  DEFINE $lemmaName;
  DEFINE $subCategory;
  DEFINE %subCategory;
  DEFINE %wordForm;
BEGIN
GetLemmaName $basicSpelling %category $lemmaName;
  IF EXISTS NODE $lemmaName %NewLemma IN @Model THEN
    RETURN -500;
  ELSE
    MakeSubCategoryName 'basic' %category $subCategory;
    IF EXISTS NODE $subCategory %subCategory IN @Model THEN
      #fastlink on;
      UPDATE @Model
        ADD NODECLASS $lemmaName %NewLemma PARENTS %category;
        AddWordForm @Model %NewLemma %subCategory $basicSpelling %wordForm;
      #fastlink off;
    ENDIF;
  ENDIF;
END AddLemma;
```

figure 27 Definition of the task *Lexicon.AddLemma*

Lexeme categories represent the forms individual lemmas of a particular category can take. For every lemma category there exists at least one lexeme category that represents the basic form of that lemma category with the appropriate feature value pairs, e.g., basic:Noun with number=singular. Additional Lexeme categories can be defined using the task *AddSubCategoryFeatures* that is presented in figure 28 below.

```
CLIENT TASK AddSubCategoryFeatures @Model $subCategory %category `features;
  // adds lexeme category and features
  DEFINE %subCategory;
  DEFINE $subCategoryName;
  DEFINE %feature;
  DEFINE %node;
BEGIN
  MakeSubCategoryName $subCategory %category $subCategoryName;
  IF NOT EXISTS NODE $subCategoryName %subCategory IN @Model THEN
    UPDATE @Model
      ADD NODECLASS $subCategoryName %subCategory PARENTS %category SubCategory;
  ENDIF;
  FOREACH NODECLASS %node IN `features DO
    IF %feature EQUALS NIL THEN
      %feature := %node;
    ELSE
      Features.AddFeatureValue @Model %subCategory %feature %node;
      %feature := NIL;
    ENDIF;
  ENDFOR;
END AddSubCategoryFeatures;
```

figure 28: Definition of AddSubCategoryFeatures

A lexeme is represented by creating an anonymous subclass of its associated lemma and its lexeme category. Its spelling is implemented by creating a *hasSpelling* link to a ROIS TEXT node that represents its spelling. The representation of lemma categories, lemmas, lexeme categories, lexemes, and spellings is illustrated in
figure 29 below. Note that the feature value pair *gender=neuter* will be inherited by all the lexemes of the noun *oor*.

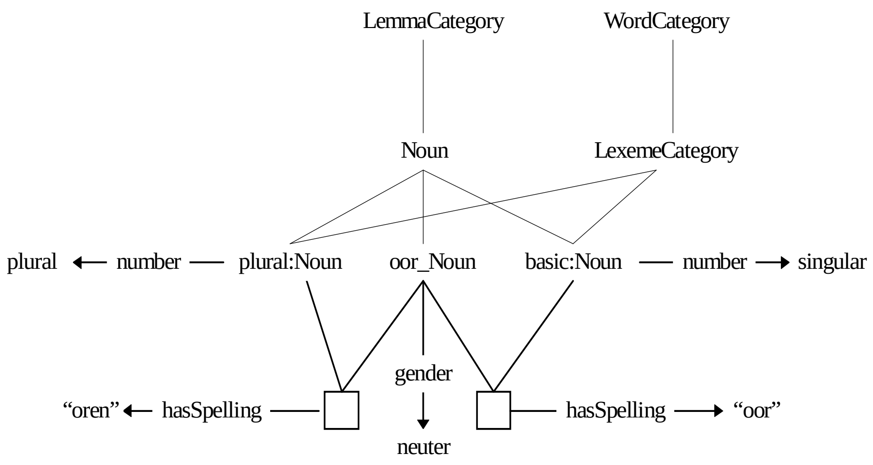


figure 29. Examples to illustrate general scheme of representation of lemmas and lexemes

The present approach allows the representation of lexemes that have the same spelling but that belong to different lemmas. For example, the Dutch string *zijn* could be the spelling of both a plural form of the Dutch verb *zijn* (*to be*) and a singular form of the possessive pronoun *zijn* (*his*). Note that this scheme also supports word forms of a single lemma that have identical spellings but different features, e.g., the masculine nominative singular and feminine genitive singular form of the German definite article (*der*). Finally, the present approach could easily be extended to represent alternative spellings and the pronunciation of lexemes.
In order to assign feature values pairs to phrase categories, lemmas, and lexeme categories, the generic model incorporates a qualifier called *actual* and a task called *Features.AddFeatureValue*.

```
UPDATE @Model
    ADD QUALIFIER 'actual'
        SANCTIONED BY potential
        PROPERTIES irreflexive
        INHERITANCE default
```

figure 30. Definition of the qualifier *actual*.

The qualifier is sanctioned by the qualifier *potential* (its definition is shown in figure 30) and assigning a feature value to a lemma or a lexeme category is implemented by creating a link of the form <phrase | lemma | lexeme category>.<feature>.actual.<feature value>.

3.4 Representation of Syntactic trees

We described syntactic tree formation as the central syntactic mechanism to be used in natural language generation and natural language analysis. In order to represent syntactic trees, the generic language model incorporates a qualifier called *constituent* which is sanctioned by the qualifier *segment*. Its definition is shown in figure 31.

```
UPDATE @Model
   ADD QUALIFIER 'constituent'
       SANCTIONED BY segment
       PROPERTIES irreflexive
       INHERITANCE no
```

figure 31 Definition of the qualifier *constituent.*

Consider the simple constituent tree representing the NP *an infection* which is presented in figure 32. The tree involves two segments: NounPhrase-head-Noun, and NounPhrase-determiner-Article.

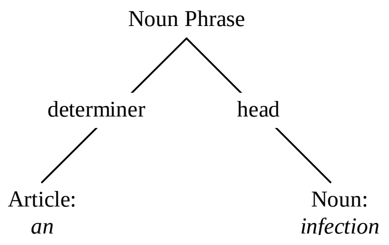


figure 32. Simple constituenty tree representing the NP *an infection*

This tree can be represented in the following way. Given an instance of the category *NounPhrase*, e.g., *[NounPhrase: #765]* and instances of the lemmas *infection_Noun* and *a_Article,* e.g., *[infection_Noun: #9876]* and *[a_Article: #6543]* we can create the following links using the link classes *NounPhrase-head-Noun* and *NounPhrase-determiner-Article*:

[NounPhrase: #765].NounPhrase-head-Noun.constituent.[infection_Noun: #9876]
[NounPhrase: #765].NounPhrase-determiner-Article.constituent.[a_Article: #6543]

Note that as soon as some linguistic expression has been produced for a certain Grail concept, the lemma and phrase instances that were used to build the syntactic tree are no longer needed and can be disposed of. For this purpose the task *Constituents.RemoveConstituent* is available.

3.5 Representation of syntactic frames

Syntactic frames are the building blocks of the natural language generation process. Examples are presented in figure 33. Syntactic frames reside in the language model. They are represented in ROIS by instances of the node class *Frame*. These frame instances can have up to four links with qualifier *internal* and respectively the link classes: *attributeSegment*, *attributeLemma*, *valueSegment*, and *valueLemma* which associate the frame instance with these segments and lemmas. For example, the simple frames in figure 33(a) and figure 33(b) only have a link to the attributeSegments (*NounPhrase-modifier-AdjectivalPhrase* and NounPhrase-*modifier-NounPhrase*). The frame in figure 33(c) has two links: one to the attributeSegment (*NounPhrase-modifier-PrepositionalNounPhrase*), and one to the attributeLemma (*of_Preposition).* The frame in figure 33(d) has four links: one to the attributeSegment (*NounPhrase-modifier-PastParticipleS*), one to the attributeLemma (*mix_MainVerb),* one to the valueSegment (*PastParticipleS-modifier-PrepositionalNounPhrase*), and one to the valueLemma (*in_Preposition*).
A frame's attributeSegment and valueSegment relate to its topic constituent and value constituent in the following way: The topic constituent of a frame always corresponds with the foot of the attributeSegment.

The value constituent of a frame corresponds either with the foot of its valueSegment (if it has one), or with the foot of the attributeSegment.

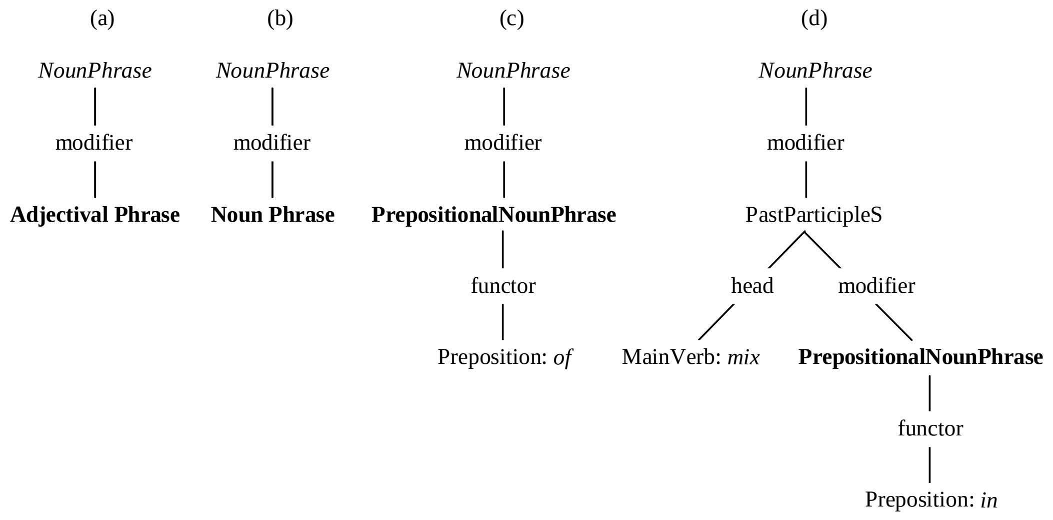


figure 33. Example frames for English. Topic constituents are in italics, value constituents in bold type face

3.6 Overview of the implementation of the generic linguistic framework

A schematic overview of the ROIS implementation of the linguistic framework is presented in figure 34. It presents the top levels of the link and node class hierarchy. Link classes are represented in boxes

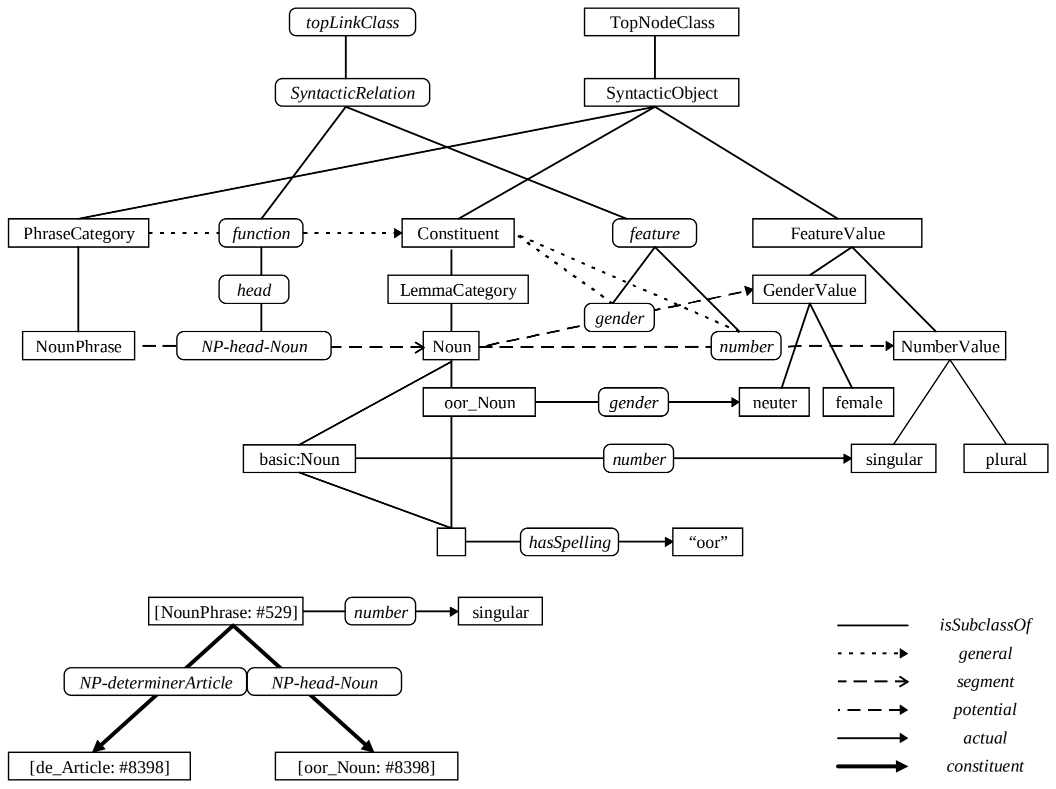


figure 34 Schematic overview of implementation of generic linguistic framework in ROIS

with rounded corners. The horizontal arrows with qualifier *general* represent the generic (language independent) sanctioning links. The horizontal links with the qualifiers *segment* and *potential* are examples of links that are part of the specification of a Dutch language model. Note that the figure is incomplete in the sense that some *potential* links are not presented for presentational reasons only (for example the link between *NounPhrase* and *NumberValue*). The links qualified as *actual* are a part of a Dutch lexicon. The links qualified as *constituent* represent the branches of an example constituent tree that would be built during the natural language generation process. Note also, that the branches are represented by links between an instance of the node class *NounPhrase* and instance of the lemmas *oor_Noun* and *de_Article*.

4 Representation of relations in Grail

In the ROIS implementation of Grail, criteria and statements are represented by links. In order to be able to annotate relations, such as the relation *(Fracture, hasLocation, Bone)* we create a link from a link that represents this relation to the node representing the syntactic frame in the language model. This is implemented as follows: first, in the Grail model a link *Fracture.hasLocation._statement_.Bone* is created using the qualifier *_statement_*. Then a hook to that link is created using the ROIS cloaking mechanism. In fact, the resulting cloak is a ROIS node that will serve as a hook to the relation *(Fracture, hasLocation, Bone)*. This is illustrated by the task addStatement from the module *System* (figure 35).

```
CLIENT TASK addStatement @model %tail %attr %qual %head :%cloak;
BEGIN
  #checking off; // allows redundant sanctions!
  IF NOT EXISTS LINK %tail.%attr.%qual.%head IN @model THEN
    UPDATE @model
      ADD LINK %tail.%attr.%qual.%head;
      ADD CLOAK %tail.%attr.%qual.%head %cloak;
  ELSIF NOT EXISTS CLOAK %tail.%attr.%qual.%head %cloak IN @model THEN
    UPDATE @model
      ADD CLOAK %tail.%attr.%qual.%head %cloak;
  ENDIF;
  #checking on;
END addStatement;
```

figure 35. Definition of task *System.addStatement*.

5 Linguistic annotation of Grail models

ROIS provides support to create links between nodes that reside in different models. This feature is used to represent linguistic annoatations of Grail models (semantic mappings from concepts of a particular Grail model to syntactic objects (lemmas and syntactic frames) of a Language model. The links that represent the semantic mappings are stored in a *semantic model*. A semantic model can incorporate links from a Grail model to several Language models. The semantic models uses the link class *hasExpression* and the qualifier *semantic* to represent the linguistic annotations. The NPL task *Semantics.createModel* creates a model that includes this link class and the qualifier *semantic*.
Before we can create a link between a concept or a relation from a Grail model and a syntactic object from a language model both the Grail model and the language model should be *added* to the semantic model using the NPL tasks *Semantics.AddGrailModel* and *Semantics.AddLanguageModel* respectively. The tasks *Semantics.EnumLanguageModels* can be used to find out which languages are associated with a particular semantic model.
Once we have a semantic model that relates a Grail model to a Language model the concept annotations can be added to a semantic model by calling the task *Semantics.AddSemanticMapping* with the concept and the lemma as arguments. This task simply adds the link if it does not yet exist (figure 36).

```
CLIENT TASK AddSemanticMapping @sem %concept %syntacticObject;
BEGIN
  IF NOT EXISTS LINK  %concept.hasExpression.semantic.%syntacticObject IN @sem THEN
    UPDATE @sem
      ADD LINK %concept.hasExpression.semantic.%syntacticObject;
  ENDIF;
END AddSemanticMapping;
```

figure 36. Definition of the NPL task *Semantics.AddSemanticMapping*

Relation annotations are represented To annotate the relation *(Fracture, hasLocation, Bone)* first the task *System.addStatement* is called with the qualifier *_statement_* from the Grail model. The hook (or *cloak* in ROIS terms) that is returned by this task can then be used as the *%concept* argument to the task *Semantics.AddSemanticMapping* shown in figure 36 above.
The argument %syntacticObject is a syntactic frame from the language model.
Concept annotations can be retrieved from the semantic model using one of the NPL tasks defined for that purpose, such as *Semantics.GetAllLemmasForEntity*. Given a particular Grail concept this task retrieves the lemmas that are capable of expressing the concept in the language represented by the input parameter *@languageModel*. The definition of this task is presented in figure 37.

```
TASK GetAllLemmasForEntity @semanticModel @languageModel %concept: `Lemmas;
  DEFINE %lemma;
  DEFINE `lemmas;
BEGIN
  FROM @semanticModel TO `Lemmas
    SELECT NODES %concept.hasExpression.semantic.?lemma
      WHERE ?lemma IN @languageModel;
END GetLemmasForEntity;
```

figure 37 Definition of the NPL task *Semantics.GetAllLemmasForEntity*

In addition to the tasks mentioned above the *Semantics* module contains several other tasks that add and remove mappings between Grail and Language models.

6 Filtering: criterion suppression and wrappers

Although it is not actually a part of the natural language generation modules, this section will describe the filtering mechanism incorporated within the ROIS Grail module. Both wrappers and criterion suppression are represented using the task *addStatement* presented in figure 35. To represent a wrapper this task is called with the qualifier *_wrapper_*, and to represent criterion suppression it is called with the qualifier *_suppress_* (see figure 38). Note that wrappers can be assigned a rank which indicates the order in which they apply.

```
CLIENT TASK addSuppress @grailModel %tail %attr %head;
  DEFINE %cloak;
BEGIN
  System.addStatement @model %tail %attr _suppress_@grailModel %head %cloak;
END addSuppress;

CLIENT TASK addWrapper @grailModel %tail %attr %head %rank;
  DEFINE %cloak;
BEGIN
  System.addStatement @model %tail %attr _wrapper_@grailModel %head %cloak;
  IF NOT %cloak EQUALS NIL THEN
    IF NOT EXISTS LINK %cloak.hasWrapperRank.systemDefinition.%rank IN @model THEN
      UPDATE @model ADD LINK %cloak.hasWrapperRank.systemDefinition.%rank;
    ENDIF;
  ENDIF;
END addWrapper;
```

figure 38 Representation of criterion suppression and wrappers

The generation algorithm calls the task Grail.definingFiltered which returns the defining criteria of a concept but which unwraps any wrappers, and which removes any criteria that are suppressed. For further detail see the sources of the Grail module (*Grail.ide* and *Filter.ide*).

7 Implementation of the generation algorithm

The implementation of the generation algorithm closely follows the description of the algorithm as presented in the part 1 of the documentation, called: *Generating Multilingual Natural Language Expressions for Grail Concepts*. The corresponding NPL code contains many comments and is largely self- explanatory. The main tasks that implement the natural language generation process are defined in the modules *Generate*, *Syntax*, and *Semantics*. Below these will be presented in turn.

7.1 Generate

```
CLIENT TASK GeneratePhrase @sem @gra @lan %concept $addArticles $phraseCategory
                           :$SurfaceString :%MainConstituent :`NoLemmaForConcept
                           :`NoSegmentForCriterion :`NoRightWordForm;
TASK GenerateConstituent @sem @lan @gra %concept %targetClass :`Constituents
                                 :`NoLemmaForConcept :`NoSegmentForCriterion;
HIDDEN TASK ExpressCriteria @sem @lan @gra %topic %topicConstituent
                            `criteria :`NoLemmaForConcept :`NoSegmentForCriterion;
```

7.2 Syntax

```
TASK FunctorizeAll @lan %const %addArticles;
TASK Serialize @lan %const :`LemmaSequence;
TASK MakeSurfaceString @lan `lemmaSequence :$Phrase :`NotRightWordForm;
```

7.3 Semantics

```
TASK GetFramesForCriterion @sem @lan @gra %concept %att %val %rootCategory :`Frames;
TASK GetLemmasForEntity @sem @lan %entity `categories :`Lemmas;
```

8 API of Natural Language Generation modules

This section lists the signatures of the tasks defined for use by clients, ordered by module.

8.1 Features

```
AddFeatureValue @Model %cat %feature %value
DeleteFeatureValue @Model %cat %feature %value
GetPotentialFeatureValues @Model %feature :`Values
GetLocalValues @Model %cat :`local
GetLocalFeatureValue @Model %cat %feature :%value
AddPotentialFeatures @Model %cat `features
GetActualFeaturesAndValues @Model %cat :`FeaturesAndValues
GetPotentialFeatures @Model %cat :`features
GetFeatureValue @Model %cat %feature :%value
ChangeFeatureValue @model %cat %feature %value
```

8.2 Lexicon

```
AddCategory @Model $name %firstParent `otherParents
ChangeSpellingOfWordForm @lan %wordForm $newSpelling
GetPotentialCategories @lan %spelling :`Categories
AddLemma @Model $basicSpelling %category :%NewLemma
GetLemmaForms @lan %lemma :`WordForms
GetFirstLemma @lan %category :%Lemma
SearchMatchingSpellings @Model $spelling :`Texts
EnumLemmaCategories @lan :`Cats
GetLemmasForEntry @lan %spelling %category :`Lemmas
AddFormFeatureValue @Model %cat %feature %value
GetNextLemma @lan %category %previousLemma :%Lemma
GetSpelling @lan %wordForm :$SpellingString
EnumSubCategories @Model %category :`SubCategories
GetLemmaForm @Model %lemma %subCategory :%Form
AddSubCategoryFeatures @Model $subCategory %category `features
GetLemma @lan $basicSpelling %category :%Lemma
AddWordForm @Model %lemma %subCategory $wordFormSpelling :%WordForm
GetFormAndSpelling @model %lemma %subCategory :%form :%spelling
HasOnlyOneSubCategory @lan %lemma :%Bool
```

8.3 Segments

```
AddSegment @Model %root %function %foot `sharedFeatures
GetRootAndFoot @lan %segment :%Root :%Foot
GetFoot @lan %segment :%Foot
GetSegment @lan %root %function %foot :`Segment
AddDestination @lan %root %function %foot %destination %position
AddSubSegment @lan %parent %root %function %foot `features
GetRoot @lan %segment :%Root
```

8.4 Constituents

```
RemoveConstituent @lan %const
```

8.5 Syntax

```
createModel $file :@Model
```

8.6 Frames

```
GetElements @lan %frame :%attributeSegment :%attributeLemma :%valueSegment
           :%valueLemma
EnumLemmaCategories @lan %segment :`Categories
EnumAttributeSegments @lan :`Segments
FindFrame @lan %attributeSegment %attributeLemma %valueSegment %valueLemma :%Frame
GetFrame @lan %attributeSegment %attributeLemma %valueSegment %valueLemma :%Frame
EnumValueSegments @lan %segment :`Segments
MakeFrame @lan %attributeSegment %attributeLemma %valueSegment %valueLemma :%Frame
```

8.7 GrailExtension

```
RemoveCriteria `criteria `minus :`Criteria
```

8.8 Semantics

```
RemoveSemanticMapping @sem %concept %syntacticObject
EnumGrailModels @model :`Models
AddGrailModel @sem $sub :@sub
AddLanguageModel @sem $sub :@sub
GetAllLemmasForEntity @sem @lan %concept :`Lemmas
EnumLanguageModels @model :`Languages
getLocalFrames @sem @language %cloak :`frames
GetEntitiesForLemma @sem @galen %lemma :`concepts
AddSemanticMapping @sem %concept %syntacticObject
makeCriterionMapping @sem @galen %topic %attr %value %segment %lemma
createModel $file :@sem
getFrames @semantics @galen @language %topic %attr %value :%statement :`frames
removeCriterionMapping @sem @galen %topic %attr %value %segment %lemma
```

8.9 Generate

```
GeneratePhrase @sem @gra @lan %concept $addArticles $phraseCategory :$SurfaceString
    :%MainConstituent :`NoLemmaForConcept :`NoSegmentForCriterion :`NoRightWordForm
```

9 External/interchange formats for Language models and Semantic models

The source files for language models and semantic models are in Lexicon Interchange Format (.LIF) and Mapping Interchange Format (.MIF) respectively. Below these formats are described in EBNF notation.

<LIF>	::=	LANGUAGE <language> <lemmas>
<language>	::=	<string>
<lemmas>	::=	1{<lemma>}
<lemma>	::=	LEMMA <lemma_category> <spelling> [<features>] [<forms>]
<lemma_category>	::=	<lemma_category_full> | <lemma_category_abbrev>
<lemma_category_full>	::=	Noun | Adjective | Article | Preposition | Adverb | ProperName | PN | CoordinatingConjunction | SubordinatingConjunction | MainVerb | AuxiliaryVerb | CopulaVerb | CardinalNumber | PersonalPronoun | PossessivePronoun | DemonstrativePronoun | InterrogativePronoun | IndefinitePronoun | ReflexivePronoun | ReciprocalPronoun | RelativePronoun |
<lemma_category_abbrev>	::=	N | ADJ | ART | PREP | ADV | COOCON | SUBCON | MV | AV | CV | CARD | PERSPRO | POSSPRO | DEMONPRO | INTERPRO | INDEFPRO | REFLPRO | RECIPRO | RELPRO
<spelling>	::=	**“**<string>**”**
<features>	::=	FEATURES 1{<feature>}
<feature>	::=	<feature_name><feature_value>
<feature_name>	::=	number | gender | definite | case | prenominal | inflection | affixRole | countable | determinable | diminutive form | tense | aspect | participle | syntacticallyTransitive | syntacticallyReflexive | reciprocal | separableVerb | diminutiveForm

```
<feature_value>          ::=   singular | plural  | masculine | feminine | neuter | + (positive) | -
                               (negative) | nominative | genitive | dative | accusative | translative |
                               partitive | essive | inessive | adessive | illative | allative | elative |
                               ablative | instructive | abessive | prefix | infix | suffix | past | present |
                               future | perfect | imperfect | presentParticiple | pastParticiple
<forms>                  ::=   FORMS 1{<form>}
<form>                   ::=   <lexeme_category> <spelling>
<lexeme_category>        ::=   “<string>”
<string>                 ::=   1{a..z | A..Z | 0..9}


<MIF>                    ::=   LANGUAGE <language> 0{<suppress>} 0{<wrapper>}
                               0{<concept_annotation >}0{<relation_ annotation >}
<suppress>               ::=   SUPPRESS <concept> <attribute> <concept>
<wrapper>                ::=   WRAPPER <concept> <attribute> <concept>
<concept>                ::=   <string>
<attribute>              ::=   <string>
<concept_annotation>     ::=   CONCEPT <concept> 1{<lemma_category> <spelling>}
<relation_annotation>    ::=   RELATION <concept> <attribute> <concept> 1{<frame>}
<frame>                  ::=   FRAME <segment> [<lemma_category> <spelling>
                                [WITH <segment> [<lemma_category> <spelling>]]]
<concept>                ::=   <string>
<segment>                ::=   <phrase_category>-<function>-<constituent_category>
<phrase_category>        ::=   NP | S | PP | ADJP | PNP | ADVP
<function>               ::=   head | modifier | functor | determiner | prefix | postfix | subject |
                               directObject | indirectObject | complement | auxiliary | particle |
                               predicate | conjunctionElement
<constituent_category>   ::=   <lemma_category_abbrev> | <phrase_category> |
                               <other_phrase_category>
<other_phrase_category>  ::=    <string>
```

Generating Multilingual Natural Language Expressions for Grail Concepts

Part III: Using Humpty and GCE

Wim Claassen
University of Nijmegen
NICI

Contents

1 Introduction 1
2 Humpty 1
2.1 Grammar specification: Humpty Grammar files 1
2.2 Compiling a grammar file 2
2.3 Adding and modifying lemmas and word forms (lexemes) 3
3 Using the GCE to add and maintain Linguistic Annotations 3
3.1 Concept and relation annotation 4
3.2 Generating language 6
3.3 Importing and exporting annotations 6
4 Adding languages to the generator 7
4.1 The Finnish, English, and Dutch language models 7
4.2 An extended example: adding German to the generator 7
4.2.1 Determine which concepts have to be described 7
4.2.2 Create a Language model 7
4.2.3 Create linguistic annotations 12
Appendix A: Grammars in Humpty format (.GRM) 14
Finnish 14
English 18
Dutch 21
Appendix B: Format of interchange files (.MIF; .LIF) 25

1 Introduction

The documentation on the ROIS based Natural Language Generator developed within the context of the Galen In Use project consist of three papers: (1) a theoretical paper on the design of the generator, (2) a description of its implementation, and (3) this paper which is a manual describing how the generator is to be used.
Grail models can be created and maintained using the ROIS client *GCE* (Galen Case Environment). Documentation of the GCE is available via the Web. The ROIS Natural Language Modules provide a collection of tasks to create grammars and lexicons for the European target languages. These tasks are called by an aplication called *Humpty*. In addition, ROIS Natural Language Modules provide tasks that support the annotation of Grail models with linguistic knowledge in order to automatically produce natural language expressions for Grail concepts. These are called by the GCE. The following sections will go into Humpty and the GCE in turn.

2 Humpty

The ROIS client application Humpty can be used first, to create and compile natural language grammars, and second, to populate and maintain lexicons that are compatible with such a grammar. Usually we refer to a compiled grammar as a *basic language model*, whereas a basic language model that has been populated with lexical material is referred to as a *language model*.

2.1 Grammar specification: Humpty Grammar files

The ROIS client application *Humpty* can be used to create basic language models. A basic language model specifies the grammar of a particular language. That is, which features potentially apply to which lemma and phrase categories, which lexeme categories are identified and what their features are, additional phrase categories that are required, and the segments that can be used to built phrases.
The syntax of grammar files is presented below in EBNF notation.

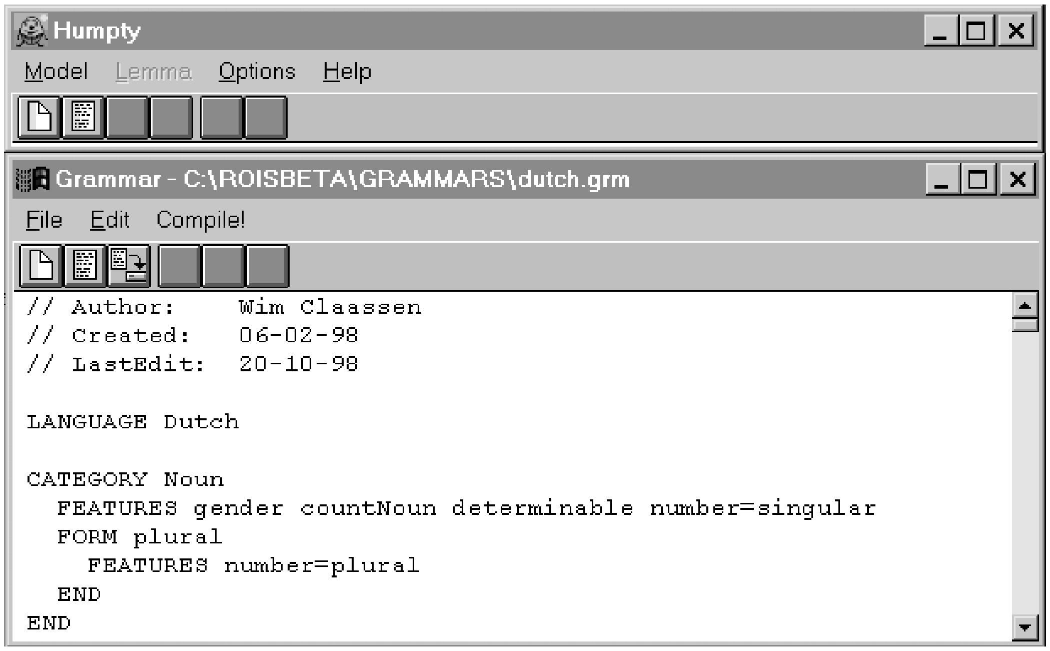
```
<grammar>                  ::=   LANGUAGE <language>
                                 0{<category_spec>}
                                 0{<segment_spec>}
                                 END <language>
<language>                 ::=   <string>
<category_spec>            ::=   <lemma_spec> | <phrase_spec | new_phrase_spec>
<lemma_spec>               ::=   CATEGORY <lemma_category>
                                 <feature_specs>
                                 0{<form_spec>}
                                 END
<phrase_spec>              ::=   CATEGORY <phrase_category>
                                 <feature_specs>
                                 END
<new_phrase_spec>          ::=   CATEGORY <string>
                                 BASE <phrase_category> | <string>
                                 < feature_specs >
                                 END
<lemma_category>           ::=   <lemma_category_full> | <lemma_category_abbrev>
<lemma_category_full>      ::=   Noun | Adjective | Article | Preposition | Adverb | ProperName | PN |
                                 CoordinatingConjunction | SubordinatingConjunction | MainVerb |
                                 AuxiliaryVerb | CopulaVerb | CardinalNumber | PersonalPronoun |
                                 PossessivePronoun | DemonstrativePronoun | InterrogativePronoun |
                                 IndefinitePronoun | ReflexivePronoun | ReciprocalPronoun | RelativePronoun
                                 |
<lemma_category_abbrev>    ::=   N | ADJ | ART | PREP | ADV | COOCON | SUBCON | MV | AV | CV |
                                 CARD | PERSPRO | POSSPRO | DEMONPRO | INTERPRO | INDEFPRO |
                                 REFLPRO | RECIPRO | RELPRO
<feature_specs>            ::=   FEATURE 1{<feature_spec>}
<feature_spec>             ::=   <feature_name> | <feature_value_spec>
<feature_name>             ::=   number | gender | definite | case | prenominal | inflection | affixRole |
                                 countable | determinable | diminutive form | tense | aspect | participle |
```

		syntacticallyTransitive | syntacticallyReflexive | reciprocal | separableVerb | diminutiveForm
<feature_value_spec>	::=	<feature_name> = <feature_value>
<feature_value>	::=	singular | plural | masculine | feminine | neuter | + | positive | - | negative | nominative | genitive | dative | accusative | translative | partitive | essive | inessive | adessive | illative | allative | elative | ablative | instructive | abessive | prefix | infix | suffix | past | present | future | perfect | imperfect | presentParticiple | pastParticiple
<form_spec>	::=	FORM <lexeme_category>[<feature_value_specs>] END
<feature_value_specs>	::=	FEATURES 1{<feature_value_spec>
<lexeme_category>	::=	"<string>"
<segment_spec>	::=	SEGMENT phrase_category> <function> <constituent_category> DESTINATION PRIMARY=<position> <shared_features> END
<phrase_category>	::=	NP | S | PP | ADJP | PNP | ADVP
<function>	::=	head | modifier | functor | determiner | prefix | postfix | subject | directObject | indirectObject | complement | auxiliary | particle | predicate | conjunctionElement
<constituent_category>	::=	<lemma_category_abbrev> | <phrase_category> | <other_phrase_category>
<other_phrase_category>	::=	<string>
<shared_features>	::=	0{FEATURES 1{<feature_name>}}
<string>	::=	1{a..z | A..Z | 0..9}
<position>	::=	1 .. 9

Example grammars in this form are presented in Appendix A.

2.2 Compiling a grammar file

When you start Humpty (e.g., by clicking the icon from the ClaW console) the Humpty console will open. If you want to create a new language model then choose *Model-New* from the menu bar. A grammar editor window will open. Here you can create and modify grammars according to the format described above (figure 39). Note that Humpty assumes that grammar files are stored in a subdirectory of your ROIS directory called *Grammars*, e.g., in C:\ROIS\GRAMMARS.



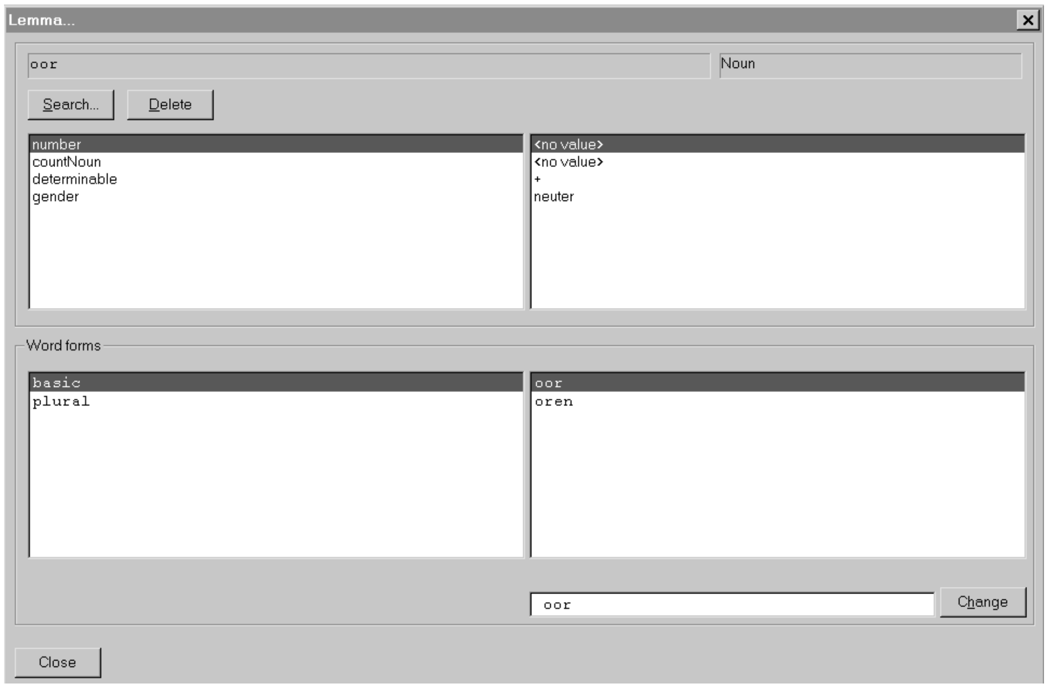
```
// Author:    Wim Claassen
// Created:   06-02-98
// LastEdit:  20-10-98

LANGUAGE Dutch

CATEGORY Noun
  FEATURES gender countNoun determinable number=singular
  FORM plural
    FEATURES number=plural
  END
END
```

figure 39: Editing grammar files using Humpty

You can choose *Compile* from the menu bar to compile the grammar into a new basic language model. Once you have succesfully compiled the grammar ,Humpty will open the new Language model for you. Then you can can close the grammar editor and start adding lemmas and lexemes to the language model.

2.3 Adding and modifying lemmas and word forms (lexemes)

In order to add or modify lemmas and word forms (lexemes) the language model must be open. You can open an existing language model by choosing *Model-Open* from the menu bar. Humpty displays the name of the open language model in its title bar. Adding lemmas and lexemes to a language model can be done in two ways, either interactively, or by importing an ascii file in .LIF format. To add lemmas interactively choose *Lemma-Add* from the menu bar. Then a window titled *Lemma* .. will open. In this *Lemma* window select the lemma category of your choice and enter the basic form of the lemma. After you press the *Ok* button the *Lemma* window will display a list of features and a list of word forms (lexeme categories) as specified by the grammar (figure 40).



oor

Noun

number	<no value>
countNoun	<no value>
determinable	+
gender	neuter

basic	oor
plural	oren

oor

figure 40: Interactively adding or modifying lemmas and word forms

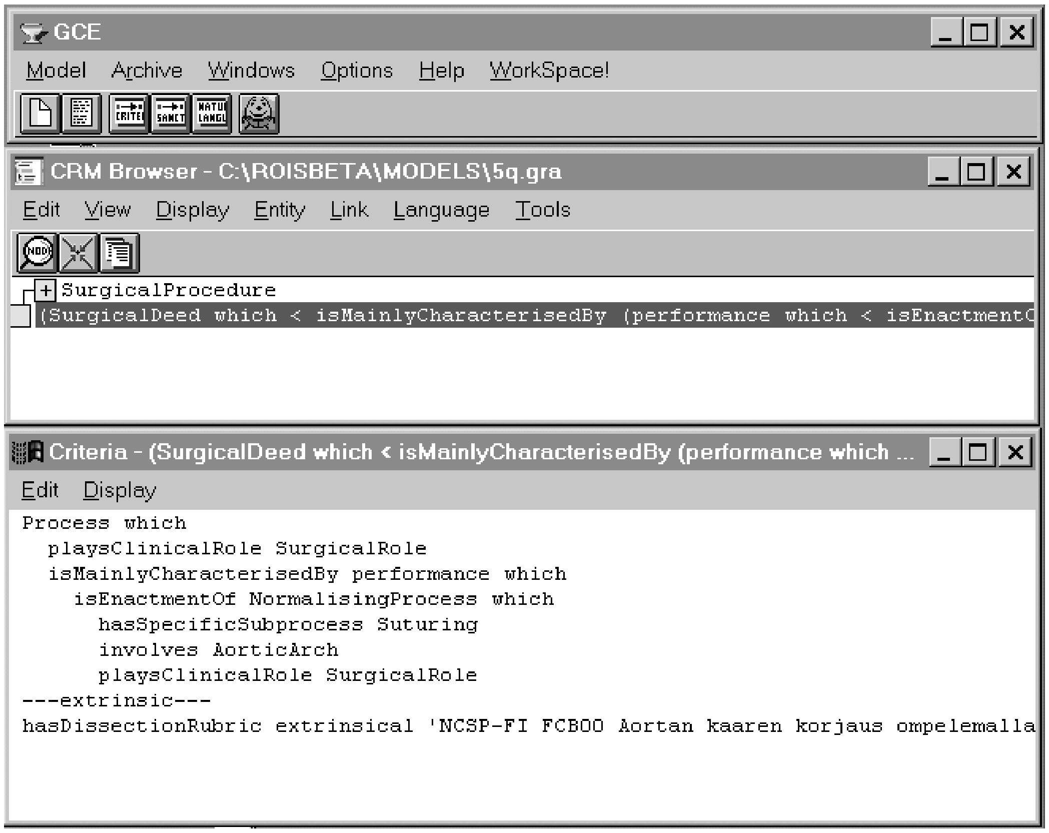
If you right click a feature value in the list, alternative values will be presented to you in a pick list. Selecting one of them will reset the feature to the new value.
You can add or edit the spelling of the individual word forms of the lemma by selecting the name of the word form (the lexeme category) and subsequently adding or modifying the contents of the text box at the bottom of the lemma window and pressing the *Change* button.
You can add lemmas and lexemes to a language model by importing an ascii file in *Lexicon Interchange Format* (.LIF; see appendix B). Then you should choose *Model-Import* from Humpty's menu bar. Conversely, you can export the lexical information from any language model to a .LIF file by choosing *Model-Export*. Note that in order to import a LIF file succesfully it must be compatible with the grammar of the language model into which it is imported. This means that the features must be applicable, and the names of the lexeme categories used must be identical. Humpty assumes that .LIF files are stored in a subdirectory of your ROIS system directory called *Sources* (e.g., in C:\ROIS\SOURCES)

3 Using the GCE to add and maintain Linguistic Annotations

You can use the GCE to add multilingual natural language generation facilities to a Grail model. To do so you must have access to a basic language model for each of the languages you want to add. To add a particular language from scratch you should create an empty MIF file for your language. This is an ascii file named <language>.MIF, (e.g., German.MIF) which resides in your ROIS\SOURCES directory. The first line of this file should be: *LANGUAGE <language>* (e.g., *LANGUAGE German*). Now first open the Grail model and a *CRM Browser* will display the top of the concept hierachy. Choose *Language-Import* from the CRM Browser's menu bar, and select the mif file from the file dialogue box that opens. Usually you will start the linguistic annotation process by tagging the wrappers and criteria to be suppressed in the Grail model. Subsequently you will add the concept and relation annotations for your language.

3.1 Concept and relation annotation

In order to add or modify annotations you should open the GCE's *Criteria* Window by choosing *Windows-Criteria* from the GCE's menu bar. Then you focus the CRM browser on the concept you are interested in. The criteria window will display the concept in canonical form (figure 41). Make sure that the option *Display-Pretty* of the criteria window is checked, and that the option *Display-Filter* of the criteria window is unchecked.

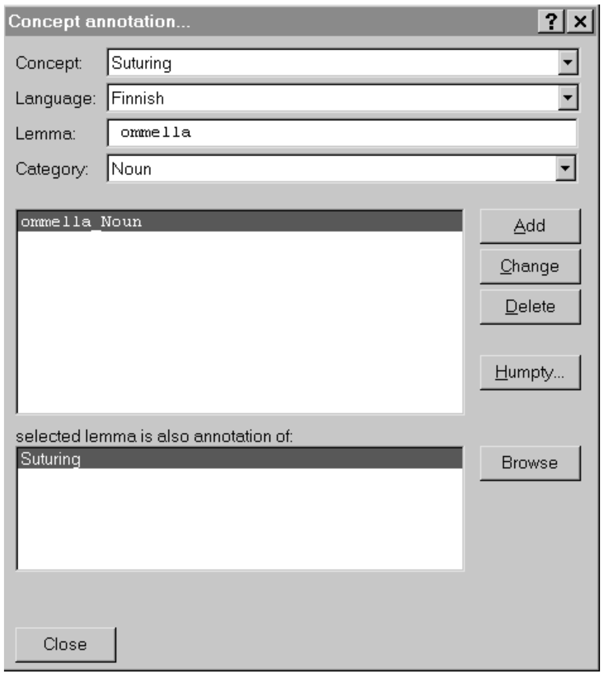

GCE
Model Archive Windows Options Help WorkSpace!
CRM Browser - C:\ROISBETA\MODELS\5q.gra
Edit View Display Entity Link Language Tools
Criteria - (SurgicalDeed which < isMainlyCharacterisedBy (performance which ...
Edit Display

```
+ SurgicalProcedure
  (SurgicalDeed which < isMainlyCharacterisedBy (performance which < isEnactmentO
```

```
Process which
  playsClinicalRole SurgicalRole
  isMainlyCharacterisedBy performance which
    isEnactmentOf NormalisingProcess which
      hasSpecificSubprocess Suturing
      involves AorticArch
      playsClinicalRole SurgicalRole
---extrinsic---
hasDissectionRubric extrinsical 'NCSP-FI FCB00 Aortan kaaren korjaus ompelemalla
```

figure 41: GCE console with CRM browser and criteria window

If you right-click on a line of the criteria window a menu pops up. You can choose *Annotate Concept* to annotate the value of the criterion that is displayed on the line clicked ,and *Annotate Statement* to create or modify an annotation for the relation that is presented on that line.
If you choose *Annotate Concept* a *Concept Annotation* window will be displayed (figure 42). It allows you to inspect, create, and modify annotations of the concept with existing or new lemmas in multiple languages using the buttons *Add, Change*, and *Delete*. You can display the lexical information on a particular lemma by selecting it from the list and pressing the button *Humpty*. You can also inspect other concepts that have been annotated with the selected lemma by selecting the concept from the bottom list and pressing the button *Browse*. This will open a CRM Browser focussed on that concept.



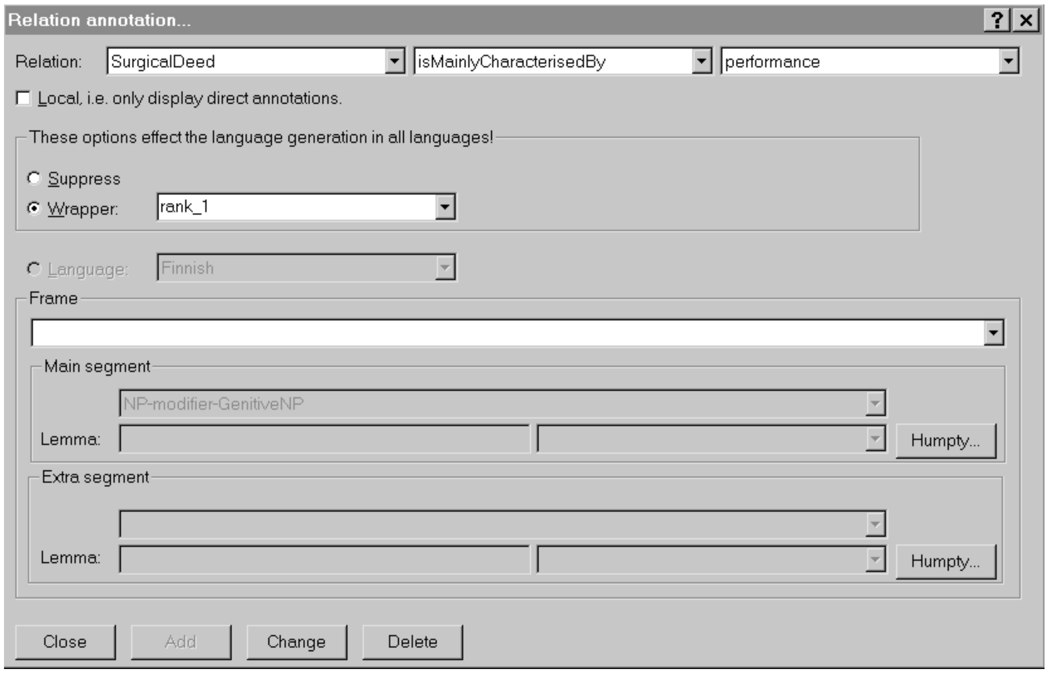
Concept: Suturing
Language: Finnish
Lemma: ommella
Category: Noun

ommella_Noun

selected lemma is also annotation of:
Suturing

figure 42: The concept annotation window.

If you choose *Annotate Statement* after right clicking on one of the lines in the criteria window the *Relation Annotation* window will be displayed (see figure 43).



Relation: SurgicalDeed | isMainlyCharacterisedBy | performance

☐ Local, i.e. only display direct annotations.

These options effect the language generation in all languages!

○ Suppress
● Wrapper: rank_1

○ Language: Finnish

Frame

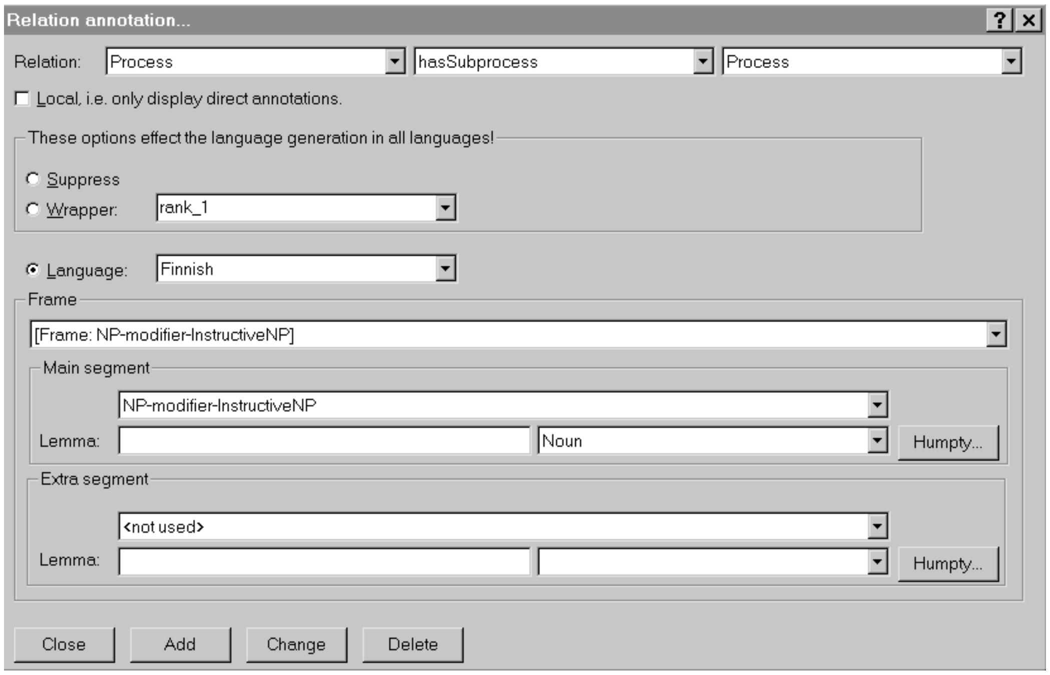
Main segment
NP-modifier-GenitiveNP
Lemma:

Extra segment
Lemma:

figure 43: Suppression and wrapper tagging using the relation annotation window.

At the top of this window the relation is presented that applies to the criterion you just right-clicked. In this example this was the criterion *isMainlyCharacterisedBy performance*. The relation presented will usually be more general than the criterion selected, that is, at a higher level in the concept hierarchy. This is so if the topic concept of the relation (*SurgicalDeed* in the example) is an ancestor of the concept focussed, if the attribute presented is an ancestor of the criterion clicked, or the value concept (*performance* in the example) is an ancestor of the value of the criterion clicked. If you want to add an

annotation at a lower level in the hierachy, you first mark the check box *Local* in order to see only relations that are local, i.e. that exist between the topic concept and the criterion value. Now you can use the drop-down lists to select more specific concepts and/or a more specific attribute.
You can suppress linguistic realisation of the current relation by clicking the radio button *Suppress.*Alternatively you can tag the relation as a wrapper by setting the rank of the wrapper using the drop-down list and pressing the button marked *Wrapper*. Please note that wrapper and suppression tagging have effects for all the languages.


Relation annotation...

Relation: Process | hasSubprocess | Process

☐ Local, i.e. only display direct annotations.

These options effect the language generation in all languages!

○ Suppress
○ Wrapper: rank_1

● Language: Finnish

Frame
[Frame: NP-modifier-InstructiveNP]

Main segment
NP-modifier-InstructiveNP
Lemma: Noun

Extra segment
<not used>
Lemma:

figure 44: Annotating a relation with a syntactic frame using the relation annotation window

To annotate the relation with a syntactic frame in a particular language, first select the language from the drop-down list (see figure 44). Then successively select the attribute (main) segment and lemma, and the value (extra) segment and lemma from the drop down lists to specify the frame that will be used to express the relation in the language selected. Note that during this process the contents of the drop-down lists may change dynamically in concordance with the segments of the grammar. Finally use the buttons at the bottom of the relation annotation window to *Add*, *Change*, or *Delete* the relation annotation.

3.2 Generating language

To see the effects of your annotations you can open the *Language* window by choosing *Window-Language* from the menu bar of the GCE console. This window will display a linguistic expression for the focussed concept of the active CRM browser. Alternatively, you can make the CRM browser display natural language instead of the concept names. To do so, choose the language of your choice from the *Display-Language* menu of a CRM browser. Note that this may take some time, depending on the number and complexity of the concepts to be displayed.

3.3 Importing and exporting annotations

Although interactively adding and modifying concept and relation annotations is useful in many contexts, it is not a very convenient way to do bulk annotations. For this purpose the concept and relation annotations can be represented in an ascii file in Mapping Interchange Format (.MIF). The format of this file is described in appendix B. To export or import your annotations respectively choose *Language-Export* or *Language-Import* from the menu bar of a CRM browser. Note that the system assumes that you store your .MIF files in the ROIS subdirectory called SOURCES.
When concept annotations contain references to lemmas that do not yet exist in the language model they will be added automatically during the language import process. It is also no problem to import a .MIF

file more than once into the same file. Mappings that are already there will be ignored by the import process.

4 *Adding languages to the generator*

In this section we will describe the steps you should take when your application needs natural language phrases to describe concepts from a Grail model. First we will say some more about the Dutch, English and Finnish grammars, lexicons, and annotations. The process of adding a language will be illustrated by an example that adds a fragment of the German language to a simple Grail model on disorders that involve the ear.

4.1 The Finnish, English, and Dutch language models

You will find the Finnish, English, and Dutch grammars in Appendix A. The Dutch and English grammars are very straightforward. Here you can see how we use Segment Grammar to create compound nouns. Using the feature *affixRole* we create the subcategories *PrefixNP*, *PrefixADJP*, *SuffixNP*, and *SuffixADJP* with the corresponding values of affixRole, and the segments where these phrase categories function as modifier, e.g., *NP-modifier-PrefixNP*. As a consequence of using the *prefix*/*suffix* feature values is that in the surface string no space is inserted before/after the head of the phrase.
You will find the corresponding Grail model, and the language models for Finnish, English, and Dutch in the software distribution. In addition you will find the corresponding LIF and MIF files to populate the language models and semantic models respectively in the subdirectory *sources*.

4.2 An extended example: adding German to the generator

Below you will find a step by step description of how to add a new language to the generator. Note that it is very useful to have a look at the example grammars, lexicons and mappings of English and Dutch before you actually start to work on your own language. It may also be very helpful to use Humpty and the GCE to look at how the other languages have been implemented.

4.2.1 Determine which concepts have to be described

Analyse the way your application uses the Grail model and create a small collection of concepts that need description in the language to be added. Especially look for concepts that are composed ‘on the fly’, as these will actually require the generator to produce ‘new’ phrases.
The example test set could consist of e.g.,

```
   (Inflammation which < hasLocation
        (Mastoid which < hasLaterality left >) hasChronicity acute >)
```

4.2.2 Create a Language model

Creating a language model involves the following steps:

1. Define the language fragment
2. Analyse the fragment
3. Create the grammar
4. Create and populate the language model

The following sections will go into each of these steps in turn

4.2.2.1 Define the language fragment

Produce a collection of example phrases

Produce a small collection of example phrases that would describe the Grail concepts in your test set in a satisfactory way. A German example phrase that could describe the example concept could be: *eine akute Entzündung des linken Mastoids.*

Word categories

Create a list of word categories that are used in the example sentences. This would produce:

Article(eine; des)
Noun (Entzündung; Mastoids)
Adjective (akute; linken)

Phrase categories

Create a list of phrase categories that play a role in the example sentences. This would produce:

NounPhrase (eine akute Entzündung ; des linken mastoids)
AdjectivalPhrase (akute; linken)

4.2.2.2 Analyse the fragment

Analyse word forms

Make an inventory of the forms of the words that play a role in the example phrases. Produce a list of alternative forms, e.g.,

Article: eine; ein einer; eines,.. der, die, das, des etc..
Adjective: akute, akut, akuter etc..
Noun: Mastoid, Mastoids, etc

Define features

This stage requires some experience, but most native speakers will be able to produce a collection of grammatical, and ungrammatical combinations of words. From this activity one can infer rules like: The form of an adjective depends on the gender and the case of the noun it modifies. The form of the article depends on the gender and the case of the noun etc. Of course, a good grammar book for your language can be very useful at this stage. These rules will help you to produce a list that associates features with word categories. E.g.,

Article:	case; gender; definite
Adjective:	case; gender
Noun:	case; gender

Define lexeme categories for each lemma category

If you think this lists of word forms and features are more or less complete define names for the lexeme categories that are needed in your fragment. Usually it is quite easy to find out what the basic form of the words of each lemma category looks like. For example, in most European languages the basic form of Nouns is the singular nominative form. Note that *basic* forms are already built into the Generic Linguistic framework, and in most languages some words (e.g., prepositions) only exist in one form wich will automatically be the basic form.

Analyze constituent structure to define segments

Now you have to analyse the example phrase with respect to its constituent structure. You should use the syntactic functions that are presented in Part I of the documentation. For the example phrase this would produce:

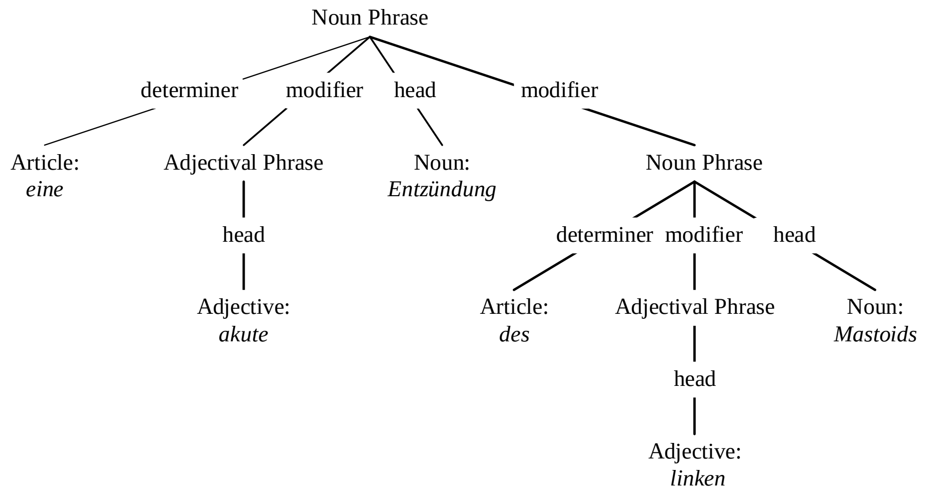


figure 45 Example of a constituent structure of *eine akute Entzündung des linken Mastoids*

From a small collection of structures like this you can infer which segments you will have to define. This would produce:

NounPhrase, determiner, Article
NounPhrase, modifier, AdjectivalPhrase
NounPhrase, head, Noun
NounPhrase, modifier, NounPhrase
AdjectivalPhrase, head, Adjective

Define phrase categories

The noun phrase *des linken Mastoids* modifies its parent phrase. It is said to be in the genitive. In order for a noun phrase to modify some other noun phrase it *must* be in the genitive form. For this reason we will later define a subcategory: *GenitiveNounPhrase* to the language model, which will have the feature case: genitive. This will allow us to substitute the segment *NounPhrase, modifier, NounPhrase* with *NounPhrase, modifier, GenitiveNounPhrase*.

Analyse how features influence the actual word form

Now we have to analyze which features of a word (lemma) should be in agreement with other constituents of the phrase. Again, a grammar handbook of your language will proof very useful at this stage. For example the features case and gender of an article should agree with the case and gender of the head of the parent constituent (the noun). We can represent this dependency by defining the shared features of the segments involved.

NounPhrase, determiner, Article:	gender; case
NounPhrase, head, Noun:	gender; case
NounPhrase, modifier, AdjectivalPhrase:	gender; case
NounPhrase, modifier, GenitiveNounPhrase:	<none>
AdjectivalPhrase, head, Adjective:	gender; case

Analyze word order in the example sentences to define segment positions

The last step in the specification of the grammar is to assign positions to segments. A position is a cardinal number which indicates the ordinal position of the foot of the segment relative to the other children of the root of that segment within the constituent structure of the phrase. When assigning positions to segments you should take into account that not all positions have to be taken by a constituent at all times. The following positions will do fine for the example fragment.

NounPhrase, determiner, Article:	second
NounPhrase, modifier, AdjectivalPhrase:	fourth

NounPhrase, head, Noun:	sixth
NounPhrase, modifier, GenitiveNounPhrase:	seventh
AdjectivalPhrase, head, Adjective:	second

Note that the first, third, fifth and eighth position for the children of a noun phrase are still unoccupied. In the extended German example grammar provided in Appendix A. You will see that they are used for other segments that have noun phrase as a root.

4.2.2.3 Create the basic language model (grammar)

Produce the grammar file in GRM format (see section 2.1).

```
// Filetype:    Grammar file (GRM)
// Author:      Wim Claassen
// File:        german.txt
// Comments:    german test grammar;

LANGUAGE German

// WORDCATEGORIES
CATEGORY Noun
  FEATURES case=nominative number=singular gender
  // (eine akute Entzündung des linken MASTOIDS)
  FORM
    "singular genitive"
    FEATURES number=singular case=genitive
  END
END

CATEGORY Article
  // only singular nominative and genitive forms, add others as required
  // definite nominative
  // der is basic form
  FEATURES definite=+ number=singular gender=masculine case=nominative
  // die
  FORM
    "definite singular feminine nominative"
    FEATURES definite=+ number=singular gender=feminine case=nominative
  END
  // das
  FORM
    "definite singular neuter nominative"
    FEATURES definite=+ number=singular gender=neuter case=nominative
  END
  // definite genitive
  // des
  FORM
    "definite singular masculine genitive"
    FEATURES definite=+ number=singular gender=masculine case=genitive
  END
  // des (eine acute Entzündung DES linken Mastoids)
  FORM
    "definite singular neuter genitive"
    FEATURES definite=+ number=singular gender=neuter case=genitive
  END
  // der
  FORM
    "definite singular neuter genitive"
    FEATURES definite=+ number=singular gender=feminine case=genitive
  END
  // indefinite nominative
  // ein
  FORM
     "indefinite singular masculine nominative"
    FEATURES definite=+ number=singular gender=neuter case=nominative
  END
  // ein
  FORM
     "indefinite singular neuter nominative"
    FEATURES definite=+ number=singular gender=neuter case=nominative
  END
  // eine (EINE acute Entzündung des linken Mastoids)
  FORM
     "indefinite singular feminine nominative"
    FEATURES definite=+ number=singular gender=neuter case=nominative
  END
  // indefinite genitive:
  // eines
  FORM
     "indefinite singular masculine genitive"
    FEATURES definite=+ number=singular gender=neuter case=genitive
  END
  // eines
  FORM
     "indefinite singular neuter genitive"
    FEATURES definite=+ number=singular gender=neuter case=genitive
  END
  // einer
  FORM
     "indefinite singular feminine genitive"
    FEATURES definite=+ number=singular gender=neuter case=genitive
  END
  // etcetera
END

CATEGORY Adjective
  // basic form = positive degree, which is not used
  // should also have feature to indicate presence/absense of article
  FEATURES definite number gender case
  FORM "definite singular masculine nominative"
    FEATURES definite=+ number=singular gender=masculine case=nominative
  END
  // (eine AKUTE Entzündung des linken Mastoids)
  FORM "definite singular feminine nominative"
    FEATURES definite=+ number=singular gender=masculine case=nominative
  END
  // (eine acute Entzündung des LINKEN Mastoids)
  FORM "definite singular neuter genitive"
    FEATURES definite=+ number=singular gender=neuter case=genitive
  END
  // etcetera (51 forms in total)
END

// PHRASE CATEGORIES
CATEGORY NounPhrase
  FEATURES gender case number definite
END

CATEGORY AdjectivePhrase
  FEATURES case gender number definite
END

///////////////////
// EXTRA CATEGORIES
//
CATEGORY GenitiveNP
  BASE NP
  FEATURES case=genitive
END

////////////////////////
// SEGMENTS
// NP
SEGMENT NP determiner ART
  FEATURES case number gender definite
  DESTINATION PRIMARY=2
END

SEGMENT NP modifier ADJP
  FEATURES case number gender definite
  DESTINATION PRIMARY=4
END

SEGMENT NP head Noun
  FEATURES gender case affixRole
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier GenitiveNP
  DESTINATION PRIMARY=7
END

// ADJP
SEGMENT ADJP head ADJ
  FEATURES gender definite affixRole
  DESTINATION PRIMARY=2
END

END German
```

4.2.2.4 Create and populate the language model

Use Humpty as described in section 2.3 to add the words from your collection of test phrases to the language model. Add the word forms and features as required.

4.2.3 Create linguistic annotations

In this step you will ‘link’ your new language model with the Grail model used by your application. First, you should tag the wrappers of your Grail model and the relations you don't want to see expressed linguistically. In our example this is not needed, but section 3.1 describes how to do this. More background information on wrapper and suppression tagging can be found in part I of the documentation. Then you should annotate the concepts from the Grail model with the lemmas from the new language, and third, you should annotate certain relations in the model with frames in your language model. Finally Below we will go into each of these activities in turn.

4.2.3.1 Concept annotations

Concept annotations are mappings from a concept in the grail model to a lemma in a language model. They can be added using GCE as described in section 3.1. The concept annotations for the example are presented below in MIF form:

```
CONCEPT Mastoid Noun "Mastoid"
CONCEPT Inflammation Noun "Entzündung"
CONCEPT left Adjective "linke"
CONCEPT acute Ajective "akut"
```

4.2.3.2 Relation annotations

Relation annotations specify how the defining criteria of a composite concepts map to syntactic frames. They are described below: In the example concept, three types of criteria are used that express localization, laterality, and chronicity respectively:

```
(Inflammation which < hasLocation
     (Mastoid which < hasLaterality left >) hasChronicity acute >)
```

You will have to decide how you want the generator to express criteria like these. In the example we chose to express the localisation of some disorder in the body using a GenitiveNp. Laterality of parts of the body and chronicity of disorders on the other hand can both be expressed by an adjectival phrase. The relation annotations for the example fragment are shown below (in MIF format).

```
RELATION BodyPart hasLaterality lateralityValueType
  FRAME NP-modifier-ADJP
RELATION Disorder hasChronicity chronicityValueType
  FRAME NP-modifier-ADJP
RELATION Disorder hasLocation BodyPart
  FRAME NP-modifier-GenitiveNP
```

criterion to lemma

The example fragment requires no frames with additional lemmas or segments. However, if we would prefer to produce the phrase *eine akute Entzündung in dem linken Mastoid* over the example phrase we would have to modify the relation annotation:

```
RELATION Disorder hasLocation BodyPart
  FRAME NP-modifier-GenitiveNP
```

to produce:

```
RELATION Disorder hasLocation BodyPart
  FRAME NP-modifier-PNP Preposition "in"
```

which means that the localization of a disorder in a body part is preferably expressed using a prepositional noun phrase in conjunction with the preposition *in*.

Appendix A: Grammars in Humpty format (.GRM)

Finnish

```
// Lines starting with two slashes are not interpreted by Humpty
//
// Filetype:    Humpty grammar (.grm)
// Author:      Wim Claassen
// File:        finnish.grm
// Comments:    Finnish grammar;
// Last Edit:   20-10-98

// indicate start of grammar file for finnish language:
LANGUAGE Finnish

// enable word affixes:
CATEGORY WordCategory
  FEATURES affixRole
END

// assign features to word categories and
// define word forms of these categories
CATEGORY Noun
  // here the statement FEATURES case=nominative number=singular
  // has two effects:
  // 1) nouns have case and number
  // 2) the basic form of noun is nominative singular
  FEATURES case=nominative number=singular
  // define additional forms:
  FORM "genitive singular"
    FEATURES case=genitive number=singular
  END
  FORM "translative singular"
    FEATURES case=translative number=singular
  END
  FORM "partitive singular"
    FEATURES case=partitive number=singular
  END
  FORM "essive singular"
    FEATURES case=essive number=singular
  END
  FORM "inessive singular"
    FEATURES case=inessive number=singular
  END
  FORM "adessive singular"
    FEATURES case=adessive number=singular
  END
  FORM "illative singular"
    FEATURES case=illative number=singular
  END
  FORM "allative singular"
    FEATURES case=allative number=singular
  END
  FORM "elative singular"
    FEATURES case=elative number=singular
  END
  FORM "ablative singular"
    FEATURES case=ablative number=singular
  END
  FORM "instructive"
    FEATURES case=instructive
  END
  FORM "abessive singular"
    FEATURES case=abessive number=singular
  END
END

CATEGORY Adjective
  FEATURES case=nominative number=singular
  FORM "genitive singular"
    FEATURES case=genitive number=singular
  END
  FORM "translative singular"
    FEATURES case=translative number=singular
  END
  FORM "partitive singular"
    FEATURES case=partitive number=singular
  END
  FORM "essive singular"
    FEATURES case=essive number=singular
  END
  FORM "inessive singular"
    FEATURES case=inessive number=singular
  END
  FORM "adessive singular"
    FEATURES case=adessive number=singular
  END
  FORM "illative singular"
    FEATURES case=illative number=singular
  END
  FORM "allative singular"
    FEATURES case=allative number=singular
  END
  FORM "elative singular"
    FEATURES case=elative number=singular
  END
  FORM "ablative singular"
    FEATURES case=ablative number=singular
  END
  FORM "instructive"
    FEATURES case=instructive
  END
  FORM "abessive singular"
    FEATURES case=abessive number=singular
  END
END

// assign features to phrase categories
CATEGORY NounPhrase
  FEATURES number case
END

CATEGORY AdjectivalPhrase
  FEATURES number case
END

// assign case to PNP and set value to partitive
CATEGORY PNP
  FEATURES case=partitive
END

///////////////////////////////////////////
// define subcategories to phrase categories

// define NominativeNP as subcategory to NP
// set case to nominative (feature case inherited from NP)
CATEGORY NominativeNP
  BASE NP
  FEATURES case=nominative
END

CATEGORY AccusativeNP
  BASE NP
  FEATURES case=accusative
END

CATEGORY GenitiveNP
  BASE NP
  FEATURES case=genitive
END

CATEGORY PartitiveNP
  BASE NP
  FEATURES case=partitive
END

CATEGORY IllativeNP
  BASE NP
  FEATURES case=illative
END

CATEGORY InessiveNP
  BASE NP
  FEATURES case=inessive
END

CATEGORY ElativeNP
```

```
  BASE NP
  FEATURES case=elative
END

CATEGORY AllativeNP
  BASE NP
  FEATURES case=allative
END

CATEGORY AdessiveNP
  BASE NP
  FEATURES case=adessive
END

CATEGORY AblativeNP
  BASE NP
  FEATURES case=ablative
END

CATEGORY TranslativeNP
  BASE NP
  FEATURES case=translative
END

CATEGORY EssiveNP
  BASE NP
  FEATURES case=essive
END

CATEGORY AbessiveNP
  BASE NP
  FEATURES case=abessive
END

CATEGORY InstructiveNP
  BASE NP
  FEATURES case=instructive
END

//////////////////////////
// define segments

// segment with root=NP, function= modifier, foot=ADJP,
// position=3, and shared features={case, number}
SEGMENT NP modifier ADJP
  DESTINATION PRIMARY=3
  FEATURES case number
END

SEGMENT NP modifier GenitiveNP
  DESTINATION PRIMARY=2
END

SEGMENT NP prefix NP
  DESTINATION PRIMARY=4
END

SEGMENT NP prefix ADJP
  DESTINATION PRIMARY=4
END

SEGMENT NP head Noun
  DESTINATION PRIMARY=5
  FEATURES case number
END

SEGMENT NP modifier PartitiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier InessiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier IllativeNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier AllativeNP
  DESTINATION PRIMARY=6
```

```
END

SEGMENT NP modifier ElativeNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier AblativeNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier AdessiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier TranslativeNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier EssiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier AbessiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier InstructiveNP
  DESTINATION PRIMARY=6
END

SEGMENT NP modifier PNP
  DESTINATION PRIMARY=6
END


// PNP inherits all of the above plus PREP:
SEGMENT PNP functor PREP
  DESTINATION PRIMARY=1
END

SEGMENT ADJP head ADJ
  DESTINATION PRIMARY=2
  FEATURES case number
END

// indicate end of grammar file
END Finnish
```

English

```
// Author:       Wim Claassen
// Last Edit:    20-10-98
// Comments:     English grammar;
// File:         english.grm

LANGUAGE English

/////////////////////
// LEXICAL CATEGORIES

CATEGORY Noun
  FEATURES countNoun determinable number=singular
END

CATEGORY Article
  FEATURES number=singular definite=positive
  FORM 'indefinite singular'
    FEATURES number=singular definite=negative
  END
  FORM 'definite plural'
    FEATURES number=plural definite=positive
  END
  FORM 'indefinite plural'
    FEATURES number=plural definite=negative
  END
END

// MainVerb
CATEGORY MainVerb
  FEATURES participle=noParticiple tense=present
  FORM 'past participle'
    FEATURES participle=pastParticiple
  END
  FORM 'present participle'
    FEATURES participle=presentParticiple
  END
END

// Coordinating and Subordinating Conjunctions
CATEGORY Conjunction
  FEATURES number determinable
END

CATEGORY CoordinatingConjunction
  FEATURES number=plural determinable=negative
END

////////////////////
// PHRASE CATEGORIES

CATEGORY NounPhrase
  FEATURES number=singular definite determinable
END

CATEGORY AdjectivalPhrase
  FEATURES number definite
END

CATEGORY NominativeNP
  BASE NP
END

CATEGORY NonNominativeNP
  BASE NP
END

CATEGORY GenitiveNP
  BASE NonNominativeNP
END

CATEGORY IndeterminableNP
  BASE NounPhrase
  FEATURES determinable=negative
END

CATEGORY RelativeS
  BASE Sentence
  FEATURES participle
END

CATEGORY PastParticipleS
  BASE RelativeS
  FEATURES participle=pastParticiple
END

CATEGORY PresentParticipleS
  BASE RelativeS
  FEATURES participle=presentParticiple
END

///////////////
// SEGMENTS

// NP
SEGMENT NP head Noun
  DESTINATION PRIMARY=6
  FEATURES number
END

SEGMENT NP determiner Article
  DESTINATION PRIMARY=2
  FEATURES number definite
END

SEGMENT NP modifier IndeterminableNP
  DESTINATION PRIMARY=5
END

SEGMENT NP modifier ADJP
  DESTINATION PRIMARY=4
END

SEGMENT NP modifier PNP
  DESTINATION PRIMARY=7
END

SEGMENT NP modifier PresentParticipleS
  DESTINATION PRIMARY=7
END

SEGMENT NP modifier PastParticipleS
  DESTINATION PRIMARY=7
END

// Conjunction is head of ConjunctiveNP so should have the same position
// as head of 'normal NP'
SEGMENT NP head Conjunction
  DESTINATION PRIMARY=6
  FEATURES number determinable
END

SEGMENT NP conjunctionElement NP
  DESTINATION PRIMARY=1
END

// PNP inherits all of the above plus PREP:
SEGMENT PNP functor Preposition
  DESTINATION PRIMARY=1
END

// ADJP
SEGMENT ADJP head Adjective
  DESTINATION PRIMARY=2
END


// S
SEGMENT PresentParticipleS head MainVerb
  DESTINATION PRIMARY=1
  FEATURES participle
END

SEGMENT PresentParticipleS directObject NP
  DESTINATION PRIMARY=2
END

SEGMENT PastParticipleS head MainVerb
```

```
  DESTINATION PRIMARY=1
  FEATURES participle
END

SEGMENT PastParticipleS modifier PNP
  DESTINATION PRIMARY=2
END

END English
```

Dutch

```
// Author:    Wim Claassen
// Created:   06-02-98
// LastEdit:  20-10-98

LANGUAGE Dutch

CATEGORY WordCategory
  FEATURES affixType
END

CATEGORY Verb
  FEATURES tense person number participle infinitiveVerb
           syntacticallyReflexive reciprocal
           separableVerb syntacticallyTransitive
END

CATEGORY MainVerb
  FEATURES number=plural tense=present
  FORM 'present singular 1'
    FEATURES number=singular tense=present person=firstPerson
  END
  FORM 'present singular 23'
    FEATURES number=singular tense=present person=secondOrThirdPerson
  END
  FORM 'past singular'
    FEATURES number=singular tense=past
  END
  FORM 'past plural'
    FEATURES number=plural tense=past
  END
  FORM 'past participle'
    FEATURES participle=pastParticiple
  END
  FORM 'present participle'
    FEATURES participle=presentParticiple
  END
END

CATEGORY Noun
  FEATURES gender countNoun determinable number=singular

  FORM plural
    FEATURES number=plural
  END
END

CATEGORY Pronoun
  FEATURES person gender number case
END

CATEGORY PersonalPronoun
  FEATURES case=nominative
  FORM nonNominative
    FEATURES case=nonNominative
  END
END

CATEGORY PossessivePronoun
  FEATURES case=nominative
  FORM nonNominative
    FEATURES case=nonNominative
  END
END

CATEGORY Article
  FEATURES gender=nonNeuter number=singular definite=positive
  FORM "definite plural"
    FEATURES definite=positive number=plural
  END
  FORM "definite singular neuter"
    FEATURES definite=positive number=singular gender=neuter
  END
  FORM "indefinite singular"
    FEATURES definite=negative number=singular
  END
  FORM "indefinite plural"
    FEATURES definite=negative number=plural
  END
END

CATEGORY Adjective
FEATURES number=singular gender=neuter definite=negative diminutive
    inflection
  FORM inflected
    FEATURES inflection=positive
  END
END

CATEGORY PhraseCategory
  FEATURES affixRole
END

CATEGORY Sentence
  FEATURES number person tense aspect mood
END

CATEGORY NounPhrase
  FEATURES number person gender definite case
END

CATEGORY AdjectivalPhrase
  FEATURES number gender definite
END

CATEGORY NominativeNP
  BASE NP
  FEATURES case=nominative
END

CATEGORY NonNominativeNP
  BASE NP
  FEATURES case=nonNominative
END

CATEGORY AccusativeNP
  BASE NonNominativeNP
  FEATURES case=accusative
END

CATEGORY DativeNP
  BASE NonNominativeNP
  FEATURES case=dative
END

CATEGORY GenitiveNP
  BASE NonNominativeNP
  FEATURES case=genitive
END

CATEGORY PrefixADJP
  BASE ADJP
  FEATURES affixRole=prefix
END

CATEGORY SuffixADJP
  BASE ADJP
  FEATURES affixRole=suffix
END

CATEGORY PrefixNP
  BASE NP
  FEATURES affixRole=prefix
END

CATEGORY SuffixNP
  BASE NP
  FEATURES affixRole=suffix
END

CATEGORY PNP
  FEATURES case=nonNominative
END

CATEGORY NP
  FEATURES number=singular
END
```

```
SEGMENT NP head Noun
  DESTINATION PRIMARY=6
  FEATURES gender affixRole
END

SEGMENT NP determiner ART
  DESTINATION PRIMARY=2
  FEATURES number gender definite
END

SEGMENT NP modifier ADJP
  DESTINATION PRIMARY=4
  FEATURES number gender definite
END

SEGMENT NP modifier PNP
  DESTINATION PRIMARY=7
END

// PNP inherits all of the above plus PREP
SEGMENT PNP functor PREP
  DESTINATION PRIMARY=1
END

//ADJP
SEGMENT ADJP head ADJ
  DESTINATION PRIMARY=2
  FEATURES gender definite affixRole
END

SEGMENT ADJP modifier PNP
  DESTINATION PRIMARY=1
END

// Morphological segments
SEGMENT NP prefix PrefixADJP
  DESTINATION PRIMARY=5
END

SEGMENT NP prefix PrefixNP
  DESTINATION PRIMARY=5
END

// S segments for simple active phrases:
SEGMENT S head MV
  DESTINATION PRIMARY=2
  FEATURES number person tense
END

SEGMENT S subject NominativeNP
  DESTINATION PRIMARY=1
  FEATURES number person
END

SEGMENT S directObject NonNominativeNP
  DESTINATION PRIMARY=4
END

SEGMENT S indirectObject NonNominativeNP
  DESTINATION PRIMARY=3
END

SEGMENT S modifier PNP
  DESTINATION PRIMARY=5
END

// Coordinating and Subordinating Conjunctions (both children of Conjunction)
CATEGORY Conjunction
  FEATURES number=plural determinable=negative
END

// Conjunction is head of ConjunctiveNP so must have the same position
// as head of NP
SEGMENT NP head Conjunction
  DESTINATION PRIMARY=6
  FEATURES number determinable
END

SEGMENT NP conjunctionElement NP
  DESTINATION PRIMARY=1
```

```
END

END Dutch
```

Appendix B: Format of interchange files (.MIF; .LIF)

The source files for language models and semantic models are in Lexicon Interchange Format (.LIF) and Mapping Interchange Format (.MIF) respectively. Below these formats are described in EBNF notation.

```
<LIF>                    ::=  LANGUAGE <language> <lemmas>
<language>               ::=  <string>
<lemmas>                 ::=  1{<lemma>}
<lemma>                  ::=  LEMMA <lemma_category> <spelling> [<features>] [<forms>]
<lemma_category>         ::=  <lemma_category_full> | <lemma_category_abbrev>
<lemma_category_full>    ::=  Noun | Adjective | Article | Preposition | Adverb | ProperName | PN |
                              CoordinatingConjunction | SubordinatingConjunction  | MainVerb |
                              AuxiliaryVerb | CopulaVerb | CardinalNumber | PersonalPronoun |
                              PossessivePronoun | DemonstrativePronoun | InterrogativePronoun  |
                              IndefinitePronoun | ReflexivePronoun | ReciprocalPronoun |
                              RelativePronoun |
<lemma_category_abbrev>  ::=  N | ADJ | ART | PREP | ADV | COOCON | SUBCON | MV | AV | CV
                              | CARD | PERSPRO | POSSPRO  | DEMONPRO | INTERPRO |
                              INDEFPRO | REFLPRO | RECIPRO | RELPRO
<spelling>               ::=  “<string>”
<features>               ::=  FEATURES 1{<feature>}
<feature>                ::=  <feature_name><feature_value>
<feature_name>           ::=  number | gender | definite | case | prenominal | inflection | affixRole |
                              countable | determinable | diminutive form | tense | aspect | participle  |
                              syntacticallyTransitive | syntacticallyReflexive | reciprocal |
                              separableVerb | diminutiveForm
<feature_value>          ::=  singular | plural  | masculine | feminine | neuter | + (positive) | -
                              (negative) | nominative | genitive | dative | accusative | translative |
                              partitive | essive | inessive | adessive | illative | allative | elative |
                              ablative | instructive | abessive | prefix | infix | suffix | past | present |
                              future | perfect | imperfect | presentParticiple | pastParticiple
<forms>                  ::=  FORMS 1{<form>}
<form>                   ::=  <lexeme_category> <spelling>
<lexeme_category>        ::=  “<string>”
<string>                 ::=  1{a..z | A..Z | 0..9}


<MIF>                    ::=  LANGUAGE <language> 0{<suppress>} 0{<wrapper>}
                              0{<concept_annotation >}0{<relation_ annotation >}
<suppress>               ::=  SUPPRESS <concept> <attribute> <concept>
<wrapper>                ::=  WRAPPER <concept> <attribute> <concept>
<concept>                ::=  <string>
<attribute>              ::=  <string>
<concept_annotation>     ::=  CONCEPT <concept> 1{<lemma_category> <spelling>}
<relation_annotation>    ::=  RELATION <concept> <attribute> <concept> 1{<frame>}
<frame>                  ::=  FRAME <segment> [<lemma_category> <spelling>
                               [WITH <segment> [<lemma_category> <spelling>]]]
<concept>                ::=  <string>
<segment>                ::=  <phrase_category>-<function>-<constituent_category>
<phrase_category>        ::=  NP | S | PP | ADJP | PNP | ADVP
<function>               ::=  head | modifier | functor | determiner | prefix | postfix | subject |
                              directObject | indirectObject | complement | auxiliary | particle |
                              predicate | conjunctionElement
<constituent_category>   ::=  <lemma_category_abbrev> | <phrase_category> |
                              <other_phrase_category>
<other_phrase_category>  ::=   <string>
```