# Development of a grammar and a graphical user interface for PETFSG

Tomas Englund
tomasen@stp.ling.uu.se

Master's Thesis in Computational Linguistics
Språkteknologiprogrammet
(Language Engineering Programme)
Uppsala University · Department of Linguistics and Philology

2005-03-21

**Supervisor:**
Mats Dahllöf, Uppsala University

**Abstract**

This Master's thesis describes the implementation of a Swedish grammar and lexicon for the grammar formalism *Prolog Embedding Type Feature Structure Grammar* (PETFSG). Previously there has only existed a grammar and lexicon for English. PETFSG is a unification based grammar formalism strongly inspired by *Head-driven Phrase Structure Grammar* (HPSG) written in Prolog by Mats Dahllöf at the Department of Linguistics and Philology at Uppsala university. The implemented grammar covers only a fragment of Swedish and the work has been aimed at particle verb constructions.

Other changes in both grammar and lexicon has been made for the adaption to Swedish.

The thesis also describes PetFsgGui, a graphical user interface that apart from making PETFSG easier to use allows the use of PETFSG without a web server. Earlier it has only been possible to use PETFSG either directly from Prolog or from a Common Gateway Interface (CGI) application which in some cases has made PETFSG rather unpractical and hard to use. This has been solved by the implementation of PetFsgGui. The PetFsgGui application is written in the program language Java.

# Contents

**Acknowledgments** **iii**

**1 Introduction** **1**
- 1.1 Purpose . . . 1
- 1.2 Outline . . . 1

**2 Head-driven Phrase Structure Grammar** **2**
- 2.1 Type hierarchy . . . 2
- 2.2 Syntactic valency . . . 2
- 2.3 Lexical rules . . . 3
- 2.4 Grammar rules . . . 3
- 2.5 Semantics . . . 4

**3 Prolog embedding typed feature structure grammar** **5**
- 3.1 Type hierarchy . . . 5
- 3.2 Named feature structures . . . 6
- 3.3 Lexical entries . . . 6
- 3.4 Lexical rules . . . 6
- 3.5 Grammar rules . . . 8
- 3.6 Morphology . . . 8

**4 Some constructions in Swedish grammar** **10**
- 4.1 Reflexives . . . 10
- 4.2 Verbs . . . 10
- 4.3 Particle verbs . . . 13
  - 4.3.1 The word order of verb particle constructions . . . 14
- 4.4 Summary . . . 15

**5 A PETFSG grammar for Swedish** **16**
- 5.1 Noun inflection . . . 16
- 5.2 Reflexives . . . 18
- 5.3 Verbs . . . 18
- 5.4 The structure of particle verb constructions . . . 20
- 5.5 Particles . . . 23
  - 5.5.1 Verb and particle . . . 23
  - 5.5.2 Verb, adverbial and object . . . 23
  - 5.5.3 Verb, particle, adverbial . . . 24
  - 5.5.4 Verb, particle, object and adverbial . . . 24
  - 5.5.5 Verb, particle, indirect object and direct object . . . 25

**6 Test suites** **26**
- 6.1 Summary of the test suite . . . 27

**7 A stand alone GUI for PETFSG** **28**
- 7.1 Overview of the programme . . . 29
- 7.2 The graphical user interface . . . 29
- 7.3 The graphical user interface (`PetFsgGui`) . . . 32
- 7.4 Interaction with Prolog (`Parser`) . . . 32
- 7.5 Details on chosen grammar elements (`Detail`) . . . 32
- 7.6 Exceptions . . . 33

7.7 Summary of the graphical user interface . . . 33

**8 Conclusions** **33**

**References** **35**

**A Grammar and lexicon** **36**
A.1 Grammar, lexicon and test suites . . . 36
A.2 The grammar . . . 36
A.3 The lexicon . . . 36
A.4 Test-suites . . . 36

**B The GUI for PETFSG** **36**
B.1 Source code . . . 36
B.2 The structure of the program . . . 36
B.3 Required software . . . 37
B.4 Required hardware . . . 37
B.5 Prerequisites for compiling the application . . . 37
B.6 Recommended structure for installation . . . 38
B.7 Compiling the program . . . 38
B.7.1 Makefile . . . 38
B.8 Running the program . . . 39
B.8.1 The script that runs the program . . . 39
B.9 A typical call to Prolog . . . 39
B.10 Java classes in the application . . . 41
B.11 Class: PetfsgGui . . . 41
B.12 Class: Parser . . . 44
B.13 Class: PrologException . . . 50
B.14 Class: Detail . . . 51

## List of Figures


1 The head-complement rule . . . . . 9
2 The lexical rule for definite forms . . . . . 17
3 The NFS for reflexive pronouns . . . . . 18
4 The FS for the reflexive pronoun *mig* (*myself*) . . . . . 18
5 The lexical rule for present tense verbs . . . . . 19
6 A feature structure that represents present tense verbs. . . . . . 20
7 The lexical rule that generates infinitival verb forms. . . . . . 21
8 The lexical rule that generates past tense verb forms. . . . . . 22
9 The NFS for verb particles . . . . . 23
10 The NFS for simple particle verbs . . . . . 24
11 The NFS for particle verbs followed by an adverbial and an object . . . . . 25
12 The NFS for a particle verb followed by its particle and an adverbial . . . . . 26
13 The NFS for a particle verb followed by its particle, an object and an adverbial . . . . . 27
14 The NFS for particle verbs followed by an indirect object and a direct object . . . . . 28
15 The structure of the program . . . . . 30
16 The main window with no grammar loaded . . . . . 30
17 Details of a NFS . . . . . 32


## List of Tables


1 The reflexives in Swedish . . . . . 10
2 The structure of sentences . . . . . 12
3 The structure of particle verbs . . . . . 14
4 The definite forms of Swedish nouns . . . . . 17

## Acknowledgments

Thanks are due to Mats Dahllöf for support and guidance. Thank you! Anneli and Pelle have also given much needed support. Thanks!

# 1 Introduction

The main purpose of this Master's thesis is to describe the implementation of a grammar for some Swedish constructions using the *Prolog Embedding Type Feature Structure Grammar* (PETFSG) formalism. The grammar and lexicon that have been available earlier describes a fragment of English and were implemented by Mats Dahllöf, the author of PETFSG, at the Department of Linguistics and Philology at the University of Uppsala.

The implemented grammar and lexicon for Swedish are restricted to some aspects of the following grammatical constructions: reflexives, verbs and verb particles.

PETFSG is a unification based grammar formalism written in Prolog by Mats Dahllöf at the Department of Linguistics and Philology at Uppsala University. PETFSG compiles grammar and lexicon files into a parser which can be accessed either directly from a Prolog system or by calling it as a *Common Gateway Interface* (CGI) application.

Semantic representations in PETFSG are expressed by the use of token dependency semantics (TDS) which is embedded in the PETFSG formalism. TDS draws inspiration from minimal recursion semantics (MRS) as described in Copestake, Flickinger and Sag (1999).

The PETFSG grammar described in this thesis is fairly close to the grammar described in Sag, Wasow and Bender (2003) with a few differences in the total number of grammatical rules used.

The thesis also describes an application which makes it possible to use PETFSG as a stand alone application with an easy to use graphical user interface, without having to use a web server. This application is written in Java and its graphical user interface (GUI) is implemented using *Java Foundation Classes* (JFC or *Swing*).

## 1.1 Purpose

The goal of the work is to write a grammar and lexicon for PETFSG which describes some aspects of Swedish. The grammar implemented by PETFSG's author is for a fragment of English. The work on both grammar and lexicon has been restricted to a partial coverage of a few grammatical constructions. The implemented constructions are outlined in the thesis.

In the course of this work it became clear that PETFSG could be adapted in order to make it easier and faster to use in the development of grammars. Hence the implementation of PetFsgGui, an application that has solved these problems. The application is written in Java using the Java Foundation Classes (*JFC* or *Swing*). The necessary connections to Prolog are handled by the classes provided in the package *Jasper* by the Swedish Institute of Computer Science (SICS). These classes are a part of the SICStus Prolog software package.

Java was considered the most suitable programming language for the application because it is both easy and powerful. Another important aspect is the availability of good classes for the construction of graphical user interfaces. Furthermore, a large number of students at the language engineering programme at Uppsala University have knowledge of Java and hopefully other persons, apart from the author of PetFsgGui, will adapt and extend its functionality when and if needed.

## 1.2 Outline

Section 2 gives an introduction to HPSG. PETFSG, for which the Swedish grammar has been written, is described in section 3.

The grammatical background is described in section 4. This section describes reflexives, verbs and verb particles in Swedish grammar. The grammatical background is summarized in section 4.4.

The implemented Swedish grammar is described in section 5. The test suites used to evaluate the implemented grammar are described in section 6.

PetFsgGui, the implemented graphical user interface for PETFSG is described in section 7 and summarized in section 7.7.

The conclusions of the thesis can be found in section 8.

Section A contains more information on the implemented grammar and lexicon. Locations where the grammar and lexicon files are specified.

Section B contains information on PetFsgGui, its source code, required software and how to compile it. A short introduction on how to write Java code that calls SICStus Prolog is also included.

# 2 Head-driven Phrase Structure Grammar

Head-driven Phrase Structure Grammar (HPSG) is a typed unification based grammar with strong lexicalism that combines elements from Generalized Phrase Structure Grammar (GPSG) and other grammar theories. HPSG was developed at Stanford's Center for the Study of Language and Information (CSLI) (Sag et al. 2003).

Both grammar and lexicon are defined by the means of feature structures (FS). The strong lexicalism means that most information is encoded in the lexicon and very few grammar rules are needed.

It is possible to incorporate semantic models into the grammatical formalism. The chosen semantic model, however, has to use FSs and unification, as these ideas are used by both grammar and lexicon.

The main idea in HPSG is the focus on the lexical heads of phrases, from which a phrase get its name and type. Furthermore it uses a sign based structure in which the linguistic knowledge is structured in a type hierarchy accordingly to their types. In HPSG the hierarchy is a direct acyclic graph, in PETFSG it is a tree. This hierarhy also allows constraint inheritance which means that daughter nodes in the hierarchy inherits features from its mother node and can be further specified.

Syntactic valency is expressed by specifying complements and specifiers in the lexical entries where wanted.

## 2.1 Type hierarchy

The elements of lexicon and grammar are assigned a type. The type expresses what features are associated to the specific element, which means that elements of the same type share the same features, but not necessarily the same values.

Elements of the same type have the same feature set and the same constraints associated to them. The different parts of speech have their own type and show their special characteristics and constraints. Grammar rules typically produce feature structures that have the type *phrase*, lexical rules are suggested to have the type *i-rule*

The types each take their place in an hierarchy, in this case a tree structure, which expresses inheritance and what features are associated with a type. This also guarantees that, for example, lexical entries of the same type has the same set of features, which is of great importance in unification. The feature set associated to a type is inherited from mother node to daughter node and it is possible to extend the feature set with appropriate features when needed in order to make the daughter node more specific.

The use of inheritance and specification makes the lexicon smaller due to the fact that the properties of the types are defined once and can be reused by simply assigning a type to a lexical entity.

## 2.2 Syntactic valency

Syntactic valency is expressed by using the features SPR (specifiers) and COMPS (complements), which are lists containing specifications of what elements are needed to make the lexical entity saturated. The elements that specify complements and specifiers are declared as feature structures. Consider, for example, the transitive verb *hit*, which needs an object upon which the act of hitting is performed. This object should be included in the list of complements (COMPS) for the lexical entry *hit*. In order to make the lexical entry grammatical a subject is also needed. This would be defined in the list of specifiers (SPR) for the lexical entry *hit*.

The implementation of HPSG as suggested by Sag et al. (2003) defines a feature *ARG-ST* (*Argument-Structure*) that is a list created by concatenating the SPR-list and the COMPS-list. This feature is supposed to be used in the creation of semantic representations of phrases and sentences. This feature is however not implemented in the available PETFSG grammars.

## 2.3 Lexical rules

Since the different word forms that can be generated from lexemes of the same category are formed in a regular way it makes sense to have some mechanism that can generate these forms automatically. This mechanism is implemented as lexical rules.

A lexical rule is a feature structure that generates word forms or lexemes from existing lexemes. This makes the writing of the lexicon less time consuming as well as the grammar files smaller in size. Although the rules are expressed as feature structures, they can be viewed as simple functions that take one argument in the form of a feature structure and returns another, the resulting, feature structure.

Since lexical rules are feature structures they should be assigned a type. Lexical rules can be of one of the following types, as suggested in Sag et al. (2003). Inflectional lexical rules generate word forms from existing lexemes, for example generating a word form for the plural form of nouns. Derivational rules on the other hand generate new word forms by the addition of a prefix or a suffix to the surface form of a lexeme. It is then possible for the generated lexeme to be transformed by other inflectional lexical rules.

Typical lexical rules are such that they generate word forms from lexemes, create plural forms for nouns or such that they generate the possible forms for the different tenses of a verb lexeme.

The feature set of lexical rules contains the following:

- The type of the lexical rule which marks the rule as either inflectional or derivational.
- The feature structure that should be altered. This feature structure has to be specific enough so that only the intended lexemes are subjected to the lexical rule, underspecified feature structures will result in over-generation of surface forms, of which many most likely will be incorrect. It should, for example, only be possible to create plural forms of nouns, not for verbs.
- The resulting feature structure. This feature structure can either be a lexeme or a word form which is shown by what type is assigned to the feature structure.

## 2.4 Grammar rules

Grammar rules govern how lexemes can be combined into phrases. The main part of the linguistic knowledge is defined in the lexicon; for example, the lexicon defines syntactic valency and what type of phrases a lexeme can modify. The rules define how the feature structures are unified. Grammar rules combine one, or more, lexemes into phrases. The rules work by unification and produce feature structures. Given the fact that the rules work in a very general way and that most linguistic information is encoded in the lexicon it follows that the number of grammar rules needed are few. The total number of grammar rules suggested by Sag et al. (2003) is six; the rules that are of most interest for the grammar fragment described in Section 5 are listed below:

- The head-specifier rule.

$$\begin{bmatrix}\text{phrase} & \\ \text{SPR} & \langle\rangle\end{bmatrix} \rightarrow \boxed{1}\ \text{H}\begin{bmatrix}\text{VAL} & \begin{bmatrix}\text{SPR} & \left\langle \boxed{1} \right\rangle \\ \text{COMPS} & \langle\rangle\end{bmatrix}\end{bmatrix}$$

This rule states that a phrase can be constructed by a lexical head and its defined specifiers.

- The head-complement rule.

$$\begin{bmatrix} \text{phrase} \\ \text{COMPS} & \langle\rangle \end{bmatrix} \rightarrow \text{H} \begin{bmatrix} \text{word} \\ \text{COMPS} & \left\langle \boxed{1},...,\boxed{n} \right\rangle \end{bmatrix} \boxed{1}...\boxed{n}$$

This rule states that a phrase can be constructed by a lexical head and its defined complements.

- The head-modifier rule.

$$\begin{bmatrix} \text{phrase} \end{bmatrix} \rightarrow \text{H}\boxed{1}\begin{bmatrix} \text{COMPS} & \langle\rangle \end{bmatrix} \begin{bmatrix} \text{COMPS} & \langle\rangle \\ \text{MOD} & \left\langle \boxed{1} \right\rangle \end{bmatrix}$$

This rule states that a lexical or phrasal head can be followed by a modifier phrase given that this phrase is declared to modify heads of the right kind.

The rules listed above are outlined in Sag et al. (2003) and are also implemented in PETFSG with a few minor changes.

## 2.5 Semantics

The semantic representation of lexemes are specified in the feature *SEM*. Sag et al. (2003) presents a simple semantic model in some detail. The semantic model in PETFSG is called token dependency semantics (TDS) and is inpired by minimal recursion semantics (MRS) and differs greatly from the semantic model described in Sag et al. (2003).

# 3 Prolog embedding typed feature structure grammar

Prolog embedding type feature structure grammar (PETFSG) (Dahllöf 2003b) is a typed unifictation based grammar formalism written in Prolog by Mats Dahllöf at the Department of Linguistics and Philology at Uppsala University. PETFSG can be used either directly from a Prolog interpretator or from a HTML-driven user interface. PETFSG currently runs on the proprietary SICStus Prolog, distributed by SICS (Swedish institute of computer science), and the free software package SWI-Prolog. SWI-Prolog is free in the sense that it is licensed under GNU GPL which gives certain rights to all users of the software. For more information on GNU GPL, see *http://www.fsf.org*.

Since the earlier implemented grammars for PETFSG have been written in HPSG inspired way, the Swedish grammar described in this thesis is also inspired by HPSG.

Semantics is expressed by the use of Token Dependency Semantics (TDS), which is inspired by MRS, described in Copestake et al. (1999). PETFSG compiles a grammar and lexicons into a parser application. Compiled applications are saved in external saved state files which can be loaded by a Prolog interpretor and reused.

A PETSG application must, at least, consist of the following parts:

- A type hierarchy.
- A grammar.
- A lexicon.
- One initial symbol.

PETFSG allows the use of lexical rules as well as named feature structures which can be thought of as defined feature descriptions that are possible to reuse in other feature descriptions. This is in some aspects different from HPSG where, for example, lexical rules are defined as feature structures. The feature descriptions used in PETFSG are possible to combine and extend as wished.

The initial symbol, of which there can be only one, defines how a well defined syntactic expression should be defined for it to be counted as a correct parse in the parsing procedure.

## 3.1 Type hierarchy

The type hierarchy in PETFSG is declared with the Prolog term `declaration`, which has the following structure:

```
declaration
TH

where

TL
```

In the first variable,*TH*, all types are declared and in the second variable, *TL*, all features that can be associated with the different types are listed. The keyword `subsume` is used to define the structure of the type hierarchy in PETFSG. The structure represents a well formed tree hierarchy in which each daughter node has only one mother.

Types can be defined by using the following:

- A name of a type declared in the type hierarchy.
- Prolog terms (which are used by their names).
- Lists which should be declared to be of the type `list`.

### 3.2 Named feature structures

Named feature structures (NFS) are descriptions of templates that are possible to reuse. They are defined using the operator `is_short_for`. NFSs have the following structure:

```
name is_short_for
    feature_structure.
```

The `name`-part can be any Prolog atom. The FS declared in `feature_structure` can use any other named feature structure as long as it has been declared before the current, which means that it has to been declared earlier in the grammar file.
The NFS for intransitive verbs has the following structure:

```
verb_intransitive is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{}].
```

In order to use a declared NFS it is necessary to list it within a FS with the keyword `nfs` before the name of the NFS. As noted above, the NFS must have been declared before any FS that uses it. Consider the following example which shows the lexical entry for the intransitive verb *le* (*smile*):

```
le >>>
   [infl_lxm,
    nfs verb_intransitive,
    lexeme: <>le].
```

This entry reuses the NFS `verb_intransitive` which declares the features and their respective values that all intransitive verbs share.

### 3.3 Lexical entries

The lexical entries of the grammar are declared in the following fashion:

```
word >>>
FS.
```

The lexical entry for the word *murmeldjur* (*groundhog*) looks like the following:

```
murmeldjur >>>
   [infl_lxm,
    nfs common_noun_spr,
    lexeme: <>murmeldjur].
```

### 3.4 Lexical rules

Lexical rules allow the generation of new lexical entries from ones already existing. They take the form of a function that operates on feature structures and generates new ones from these. To each lexical rule a morphological rule is connected which transforms the surface form of the generated lexeme or word form. It is however possible to override morphological rules for lexemes that do not adhere to the regular way of forming the different possible forms. It is also possible to connect a Prolog call to a lexical rule

which will be called when the lexical rule is executed. All defined lexical rules are executed when a PETFSG application is compiled. The form of lexical rules are as follows:

```
lexrule name
morph name_of_morphological_transformation
input feature_structure
output feature_structure
prolog_call
```

It is possible to choose any name for a lexical rule, even though it is recommended to choose a descriptive name. The name of the lexical rule has to be a Prolog atom for the lexical rule to be parsed by PETFSG. A morphological rule that is connected to a lexical rule must be defined, otherwise the lexical rule will not be parsed, i.e not added to the application. For the syntax of morphological rules, see Section 3.6. In the feature structure descriptions that make up the parts `input` and `output` it is possible to use variables which make the feature structures more general.
As an example, consider the following lexical rule which creates the plural forms of nouns:

```
lexrule plur_noun
  morph  plur_noun_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                cnnoun],
          spr:x3,
          comps:x4,
          slash:x5]
  output [word_valency,
          token:x0,
          lexeme:x1,
          head:[x2,
                cnnoun,
                agr:[num:plur]],
          spr:x3,
          comps:x4,
          slash:x5].
```

## 3.5 Grammar rules

Grammar rules are defined using the following syntax:

```
name
  RESULT
===>
   FSs
```

Where `name` is a string that describes the nature of the rule. The name of the rule has to be a Prolog atom for the rule to be recognized by PETFSG. The feature structure is the resulting phrase after the rule has been applied. The variable contains the necessary components for the rule to be satisfied. This variable can contain the following:

- Feature structures that represent the minimal set of elements and values needed for the rule to be satisfied. This is expressed using features and values.

The following example shows the head-complement rule, which combines a lexical head with its first missing complement. The rule can be applied several times in order for a lexical head to be combined with all its missing complements.

```
hd_comps rule

[phrase_hc,
 token: <>(T1+T2),
 head:xHFP,
 comps:xComps,
 spr:xSpr,
 pm:neg_p]                          ===>

[  [token: <>T1,
    head:xHFP,
    spr:xSpr,
    comps:xComp^xComps,,
    pm:neg_p],

   [xComp,
    token: <>T2,
    pm:neg_p]].
```

The rule above shows how grammar rules are written in the grammar file, Figure 1 shows the rule in the format PETFSG displays rules to the user, i.e HTML

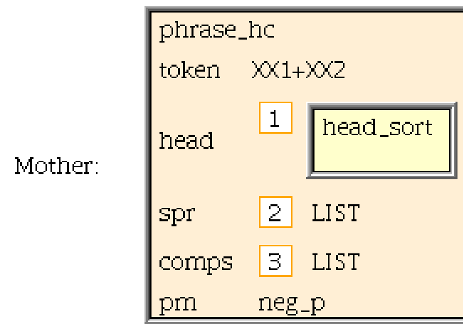
## 3.6 Morphology

Morphology in PETFSG is handled by a combination of lexical rules and Prolog predicates that alter the surface form of a given lexeme. Lexical rules change the values of the features associated with a lexeme to values connected to certain forms of a lexeme. The morphology process in PETFSG is handled by general Prolog computation. This makes it possible to alter the surface form of a lexeme and features associated with it. It is also possible for morphological rules to not change the surface form which is useful for lexemes that do not adhere to the regular way of forming certain word forms. This is done by using the predicate `suppletive_form`.

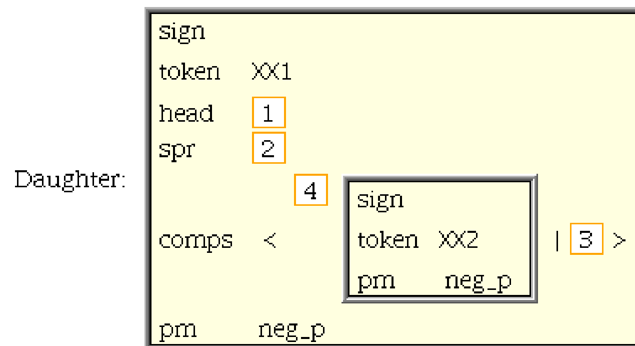
Morphological rules have to be connected to a lexical rule, as shown in Section 3.4.

PS rule **hd_comps**

Mother:




Daughter:




Daughter: 4

Figure 1: The head-complement rule

The morphological function associated with a lexical rule has to be connected with a Prolog predicate that either adds a suffix to a surface form or leaves it as it is. This association is made with the Prolog predicate `infl_reg/2`. The following predicate states that the suffix `ar` should be added to nouns (in Swedish) in their plural form.

```
infl_reg(plur_noun_infl,affx(ar)).
```

The first argument is the name of the morphological rule, which also is declared in the lexical rule, and the second argument is a call to a predicate that changes the surface form of the lexeme.

For nouns that do not adhere to this rule, for example the noun *bokhandel* (*bookstore*), the following has to be declared.

```
suppletive_form(bokhandel,plur_noun_infl,boklådor).
```

The first argument of the call is the lexeme whose morphological reshaping should be overridden, the second argument states which morphological function it is that should be overridden and the third argument is the surface form that should be given to the lexeme instead of that that would have been generated by the defined rule. Without this declaration the plural form of *bokhandel* (*bookstore*) would be **bokhandelar* (*bookstores*).

# 4 Some constructions in Swedish grammar

This section will describe the grammatical background that has been taken in account when implementing the Swedish grammar.

## 4.1 Reflexives

Reflexive pronouns normally co-refer to the subject in a phrase and are placed in the position of the object. In Swedish there is agreement between the reflexive and the subject to which it refer. The reflexive pronouns in Swedish are shown in table 1.

| **number** | **1st person** | **2nd person** | **3rd person** |
|---|---|---|---|
| **singular** | mig<br>*myself* | dig<br>*yourself* | sig<br>*itself* |
| **plural** | oss<br>*ourselves* | er<br>*themselves* | sig<br>*themselves* |
| **singular reflexive possessive pronoun** | | | sin sitt sina<br>*its its its* |
| **plural reflexive possessive pronoun** | | | sin sitt sina<br>*its its its* |

Table 1: The reflexives in Swedish

## 4.2 Verbs

Verb forms belong to one of the two classes finite and non-finite verb forms. It is only possible for finite verb forms to act as predicates in sentences. Non-finite verb forms do not themselves contain information about tense and express tense with the aid of auxiliary verbs.

Verb phrases consist of at least one or more auxiliary verbs and one finite verb (Thorell 1977). The possible verb phrases in Swedish as suggested by Thorell (1977) are shown below.

- *Auxiliary verb(s) + infinitive verb*
  The auxiliary verbs are used to express mode, tense or how the action described by the verb is performed.

  The infinitive verb can be constructed either by using the infinitive marker *att* (*to*) or without using it. As an example consider the sentence *Jag vill sova* (*I would like to sleep*). As an example of a verb phrase using the infinitive marker consider the following sentence *Jag kommer att befinna mig vid datorn* (*I will be by the computer*).

- *Auxiliary verb + part participle verb*
  In phrases like this the auxiliary verb used is *ha/har/hade* (*have/having/had*). The auxiliary verb can be excluded in subordinate clauses.

- *Auxiliary verb + present participle verb*
  Consider the following examples taken from Thorell (1977), *Bilen blev liggande på taken* (*The car remained lying on its roof*) or *Hon ansågs vara vållande till olyckan* (*She was considered to be responsible for the accident*).

- *Auxiliary verb + past participle verb*
  The following examples from Thorell (1977) illustrates this type of verb phrase, *Hon är beundrad* (*She is admired*) and *Han ville inte ha betalt för hjälpen* (*He did not want to be payed for the help*).

- *Verb + reflexive pronoun*
  The sentences *Hon anmälde sig till polisen* (*She reported herself to the police*) and *Han ångrar sig* (*He regrets something*) illustrates this kind of verb phrase.

  Many verbs can only be used together with a reflexive pronoun.

- *Verb + prepositional phrase*
  The following sentences illustrates this type of verb phrase, *Han står på stolen* (*He stands on the chair*) or *Mitt tålamod sätts på prov* (*My patience is tested*).

  The verb preceding the prepositional phrase is often of the character that it shows changes leading towards a certain goal. It also often marks a change. The most commonly used prepositions are *i* (*in*), *på* (*on*) and *till* (*to*).

  Many verb phrases of this type is lexicalized phrases and the nouns of the preposition phrase does not occur in the given form apart from in the verb phrase. Consider the examples *Något är i görningen* (*Something is in the making*) and *Gå till väga* (*Set about something*).

The different particle verb constructions consisting of a verb and a particle in different combinations are listed in table 3.

The main type of different sentences at hand are:

1. Active sentences

2. Passive sentences

3. Interrogative sentences

Table 2 derivated from Holm and Larsson (1976) shows the structure of the different type of sentences. The different columns in the table refers to what elements that normally are placed in them. Note that it is possible to put phrases in the columns, not just single words.

The elements in the first row of the table enclosed within parenthesis are optional and can be left out of a sentence. Examples (1)-(5) in table 2 shows variations on the same sentence in which different elements of the sentence is made fundament. The element that is made fundament is chosen in a way such that the focus of the sentence is set to a specific part of it. Sentence (6) in table 2 shows an active sentence in which the subject is made fundament. The finite verb takes its place on the second position. In interrogative sentences as (5) in table 2 the subject takes its place directly after the finite verb. Passive sentences, as examples (7) and (8) in table 2 show can be created in two ways. In example (7) the finite verb has been changed to the verb's passive voice. This verb form is created by adding the suffix **s** to the verb. The subject and the object also changes position with each other. Furthermore the subject is made into a preposition phrase consisting of the preposition *av* (*by*) and the former subject. Passive sentences can also be formed as in example (8) in which the subject and object changes places with each other. The verb is replaced with its past participle form preceded by the verb *bli* (*is*). It is also the case here that the subject is turned into a preposition phrase consisting of the preposition *av* (*by*) followed by the subject. According to Holm and Larsson (1976) the sentences (1)-(8) in table 2 form a structure that is almost always followed in Swedish. There are however exceptions in which parts of a sentence can occur in places not really covered by the current schematics. Sentences (9)-(17) show such exceptions. In sentence (9) an adverbial *snart* (*soon*) is placed directly before the subject. This can be done for adverbials that are not stressed. An adverbial that is not stressed can also be placed directly before the second predicate, as in sentence (10). Adverbial phrases normally placed in the field *Adv 2* can be moved to the field preceding the second predicate. Sentences written in this style are less casual than ones in which the adverbial phrase is placed in the field *Adv 2*. In examples (11) and (12) adverbials usually placed in the field of adv 2 have been placed in the field of adv 1. Objects that are pronouns can be placed before the first adverbial as in example (13). This can only be done if the object is not stressed and the sentence is written in either present or past tense. Negated objects can be placed in the field ADV 1 when

| Number | Fundament | Pred 1 | (Subj) | (Adv 1) | (Pred 2) | (OBJ/PF) | (Adv 2) |
|---|---|---|---|---|---|---|---|
| 1 | Män<br>*Men* | måste<br>*must* | | alltid<br>*always* | ha<br>*wear* | kostym<br>*a suit* | på bröllop<br>*at weddings* |
| 2 | Alltid<br>*Always* | måste<br>*must* | män<br>*men* | | ha<br>*wear* | kostym<br>*a suit* | på bröllop<br>*at weddings* |
| 3 | Kostym<br>*A suit* | måste<br>*must* | män<br>*men* | alltid<br>*always* | ha<br>*wear* | | på bröllop<br>*at weddings* |
| 4 | På bröllop<br>*At weddings* | måste<br>*must* | män<br>*men* | alltid<br>*always* | ha<br>*wear* | kostym<br>*a suit* | |
| 5 | | Måste<br>*Must* | män<br>*men* | alltid<br>*always* | ha<br>*wear* | kostym<br>*a suit* | på bröllop?<br>*at weddings?* |
| 6 | En kvinna<br>*A woman* | stänger<br>*closes* | | | | fönstret<br>*the window* | |
| 7 | Fönstret<br>*The window* | stängdes<br>*was closed* | | | | | (av en kvinna)<br>*(by a woman)* |
| 8 | Fönstret<br>*The window* | blev<br>*was* | | | stängt<br>*closed* | | (av en kvinna)<br>*(by a woman)* |
| 9 | Nu<br>*Now* | börjar<br>*starts* | snart<br>*soon* | programmet<br>*the show* | | | |
| 10 | Programmet<br>*The show* | har<br>*has* | | redan<br>*already* | visats<br>*been shown* | | |
| 11 | Henne<br>*Her* | lät<br>*had* | han<br>*he* | sedan<br>*later* | avskeda<br>*fired* | | |
| 12 | Har<br>*Has* | han<br>*he* | på grund av ...<br>*on account of ...* | missat<br>*missed* | labben<br>*the lab* | | |
| 13 | Känner<br>*Know* | du honom<br>*you him* | inte<br>*not* | | | | |
| 14 | Ville<br>*Wanted* | han<br>*he* | inga pengar<br>*no money* | ha<br>*have* | | | |
| 15 | Det<br>*There* | visas<br>*shown* | | aldrig<br>*never* | | bra program<br>*good shows* | på TV<br>*on TV* |
| 16 | Sover<br>*Sleeps* | gör<br>*do* | jag<br>*I* | aldrig<br>*never* | | | |
| 17 | Sovit<br>*Slept* | har<br>*have I* | jag<br>*never* | aldrig<br>*(done)* | (gjort) | | |

Table 2: The structure of sentences

both PRED fields are filled. If not both PRED fields are filled this can not be done. Example (14) shows this structure. Example (15) shows a sentence in which the subject occurs twice. The formal subject *det* (*it*) takes the position of the fundament whereas the formal subject is placed in the field OBJ/PF. The formal subject *det* (*it*) can be excluded in sentences that starts with an adverbial phrase. The finite verb can as in example (16) and (17) be made fundament. In such sentences the first predicate has to be a form of *göra* (*does*) otherwise the sentence will be interrogative.

As opposed to verbs in English, Swedish verbs do not show agreement with the subject in sentences. This means that a verb in any of its forms and tenses has the same form regardless of person and number of the subject.

Finite verbs are inflected with regards to tense and mode. Swedish has two simple tenses, present and past. These are formed either by altering the verb by the adding of a suffix to it or altering its stem. Verbs either have strong or weak inflections depending on how its different forms are created. A verb that forms an inflection by adding a suffix is said to be of weak inflection, otherwise it is of strong inflection. The past tense of verbs of the weak inflection is usually formed by the addition of the suffix `de`, *hitta - hittade* (*find - found*), the suffix `-te` is used when the verbs end with a toneless consonant (Östen Dahl 1982). Verbs of strong inflections change their stem in the forming of the different tenses. An example of this is the verb `skjuta` (*shoot*) whose past tense is *sköt* (*shot*).

Future tense of verbs normally has to be formed with the aid of auxiliary verbs. Typically in the following fashion: `kommer att (will) + infinitival verb form`, the future tense is also frequently formed as: `ska (shall)+ infinitival verb form` (Östen Dahl 1982).

Except in interrogative sentences, predicates are always placed on the second position, after the subject, in Swedish sentences (Holm and Larsson 1976). In interrogative sentences the predicate is always placed first.

## 4.3 Particle verbs

Particle verbs are lexical and semantical constructions consisting of a verb and a particle. A particle is normally stressed and the verb to which it belongs is not. The particle and the verb are semantically and positionally linked together. A verb can at most have one particle.

In some cases the particle appears as a prefix to the verb and in other cases the particle is separated from the verb and appears after it. Particle verbs of the latter type are said to be a loosely connected and those of the former are said to be firmly connected. The meaning of verbs that are firmly and loosely connected are sometimes different. The implemented grammar that is described in this thesis deals only with verb particles of the loosely connected kind.

Particle verbs with a loosely connected particle form a single semantic unit, even though they can be separated by other words.

The fundament of a sentence is placed first in a sentence and marks the theme of the sentence in regards to what the speaker/writer wants to be the starting-point or the focus of a sentence (Holm and Larsson 1976). The fundament can also function as a way to introduce a semantic structure that can be elaborated or explained further on in the sentence. For example consider the following sentences: *Han älskar verkligen sin dotter* (*He really loves his daughter*) in which *Han* (*He*) is the fundament of the sentence. The fundament of the sentence can also be changed as in the sentence *Sin dotter älskar han verkligen* (*His daugher he really loves*) (The example is taken from Holm and Larsson (1976)). In declarative main clauses particles can not be made to be fundament, for example compare the sentence *Jag hittar på* (*?I make up*) to **På hittar jag* (**Up I make*).

Particles are often similar to adverbs as in *hälla upp* (*pour*) or *slå ner* (*˜knock s.b down*) that marks direction or prepositions such as in *gå av* (*break*) or *hitta på* (*make up*). Particles can also be similar to nouns as in *slå vakt* (*guard*), adjectives *bryta löst* (*break loose*) or even verbs in participle form, for example *göra gällande* (*˜make a statement*). Even though they may appear to be of a number of different parts of speech particles should be marked as "miscellaneous adverbs" according to Teleman (1974).

The most common place for the particle is directly after the verb to which it belongs and before any other complements and modifiers of the verb. For some verb particle constructions the verb and the particle can be separated by an object, especially if the object is reflexive, for example *ta sig an någonting* (*set about something*), *bära sig åt* (*behave*) or *göra sig av med någonting* (*get rid of something*). There are however cases in which objects that are not reflexives can be placed between the verb and the particle as in the phrases *skilja dem åt* (*separate them*) or *stå någon bi* (*stand by someone*).

An adverbial that marks location can be separated from a particle by looking at its location in the sentence. Adverbials that marks location are adverbials if they appear after the object (Teleman 1974).

In speech the particle is always stressed, a fact that can be used to distinguish particles from other parts of speech.

An analysis of how to separate a preposition from a particle is outlined below. According to Teleman (1974) it is possible to distinguish particles from prepositions using the following criteria:

1. If an indirect object appears before a direct object as in the sentence:
   *ta ifrån flickan hennes godis* (*take the candy from the girl*)
   the preposition should be interpreted as being a particle.

2. If the supposed preposition can be made a prefix of the verb, without altering the word order any more it is to be interpreted as a particle. For example, consider the following:
   *ta (i)från flickan hennes godis* (*take the candy from the girl*)
   *frånta flickan hennes godis* (*take the candy from the girl*)

3. If the supposed preposition together with a following nominal phrase can be made fundament or first element of a relative clause or a interrogative clause it is to be interpreted as a preposition. Example:
   *Flickan håller i godispåsen* (*The girl holds the bag of candy*)
   *Det är i godispåsen flickan håller* (*It is the bag of candy that the girl holds*)

   This holds for sentences in which the particle has weak intonation.

4. In cases in which the nominal phrase following the supposed preposition can be made subject of a passive phrase it is to be intepreted as a particle.

5. In cases where the nomimal part of a prepositional phrase is emphasized by the use of the word *särskilt* (*especially*), it is possible to place the word *särskilt* (*especially* before prepositions with the intent of marking the following object as in the sentence *jag har lekt särskilt med Kalle* (*I have played especially with Kalle*). It is however not possible to place the word *särskilt* before a particle in order to mark the object as in **jag har brutit särskilt av käpparna* ( *I have especially broken the sticks*).

#### 4.3.1 The word order of verb particle constructions

Teleman, Hellberg and Andersson (1999) list the word order for Swedish verb particle constructions as shown in table 3. The structures outlined in this table are the basis for the implemented grammar, as far as the implementation of particle verbs goes.

| Number | Verb | Reflexive/Adv. phrase | Particle | Adverbial | Object | Object | Adverbial |
|---|---|---|---|---|---|---|---|
| 1 | gå | - | av | - | - | - | - |
| 2 | göra | - | bort | - | sig | - | - |
| 3 | slå | - | av | på takten | - | - | - |
| 4 | sikta | - | in | - | sig | - | på någonting |
| 5 | ta | - | ifrån | - | eleven | pennan | - |
| 6 | ta | sig | an | - | någonting | - | - |
| 7 | gå | snabbt | åt | - | - | - | - |

Table 3: The structure of particle verbs

The simplest word order for verb particle constructions is shown in example (1) in table 3, namely a verb followed by a particle. The particle is normally placed directly after the infinite verb and before any other complements to the verb, as mentioned earlier can a verb at most have one particle connected to it.

Example (2) in table 3 shows a simple verb particle construction followed by a reflexive object, this structure is close to that shown in example (1) in table 3.

In some verb particle constructions prepositional phrases can function as particles, as in the following examples *Han satte i gång diskmaskinen* (*He started the dishwasher*) or *Han körde i väg nasaren* (*He made the monger go away*). For prepositional phrases that can only function as particles it is often the case that they are lexicalized phrases, such as *i ordning* (*in order*), *(ta) till vara* ( *make the most of*) or *i land* (*on land*). Such prepositional phrases are, according to Teleman et al. (1999), to be viewed as adverbials. This structure is shown in example (3) in table 3.

The structure shown in example (4) in table 3 shows a verb particle construction in which the particle is followed by a prepositional phrase, functioning as an adverbial.

Example (5) in table 3 shows a verb particle construction in which there are two objects, an indirect and one direct. In constructions like this the particle is most often a preposition. It is rare for adverbs to be followed by an object and an objectival predicative, for example **Han gjorde om garaget lite lägre* (**He remade the garage a bit lower*). Some constructions of this form can be paraphrased so that the object is placed on the place of the particle, *ta med sig någonting* (*bring something*) can be written as *ta någonting med sig* (*bring something*). However, this does not hold true for all constructions, for example *ha för sig någonting* (*imagine something*) does not mean the same thing as *ha någonting för sig* (*be up to something*).

Verb particle constructions with an object and an adverbial, such as (4) in table 3, can not be paraphrased as a construction with two objects if the parts of speech that the adverbial governs not can be seen as the receiver of the action. Compare *Han siktar in sig på en seger* (*He aims for a victory*) to **Han på en seger siktar in sig* (**He for a victory aims*).

As mentioned earlier there are alternative structures for verb particles in which the particle is not placed directly after the verb. In some particle verb constructions the verb and the particle are separated by a reflexive pronoun, for example *ta sig an någonting* (*take up something*) or *bära sig åt* (*behave*). This type of construction is listed as example (6) in table 3. Constructions like this are often frequent when the verb is also followed by an object that could be viewed as a complement of the preposition, as in the following phrase *Hon lägger sig i saker* (*She interferes*). Many verbs have a reflexive pronoun in the place of the object, as in *äta upp sig* (*gain weight*) or *tråka ut sig* (*bore oneself*). For some verbs it is possible to place the reflexive pronoun either directly after the verb and before the particle or after the particle. The meanings of such phrases are however not necessary the same, compare *ställa sig in hos någon* (*curry favour with somebody*)to *ställa in sig hos någon* (*report to somebody*).

It is possible for an adjective or an adverbal phrase to be placed between the verb and the particle, as in *Han skämmer snabbt ut sig* (*He quickly puts himself to shame*), *Hon skällde mycket på honom* (*She scolded him a lot*) or *se sliten ut* ( *look worse for wear*). This word order is listed in example (7) in table 3. The verbs in such phrases normally have weak intonation and the adjective or adverbial phrase has as strong intonation as the particle or even stronger.

For a small number of verb particle constructions, an adverb or preposition that together with the verb is part of a lexicalized phrase is separated by an object. These adverbs or prepositions are however to be viewed as particles. Examples of such constructions are *Vi får väl se tiden an* (*Time will tell*) or *Det brukar höra henne till att tacka för maten* ( *She normally says thanks for the dinner*).

For declarative main clauses, particles can normally not be made fundament of a sentence. Consider the following examples: **på hittar jag* (**Up I make*) or **Ut skämde hon sig* (*?To shame she put herself*). However, for intransitive verbs can some particles be fundaments, as in *In kom nu en stor man* (*A big man now entered the room*) or *Ut kommer de aldrig* (**Out they will never come*).

Example (6) in table 3 shows the use of a reflexive pronoun as a reflexive object. In such cases the verb and the reflexive pronoun are often lexicalized units. Further examples of such constructions are *bita sig fast* (*cling tight to something*)), *arbeta sig upp* (*work one's way up*) and *bära sig åt* (*behave*).

The structures in table 3 is the basis of the implemented grammar.

## 4.4 Summary

The discussed grammatical aspects discussed in this section will be fundation of the implemented grammar as described in section 5.

# 5 A PETFSG grammar for Swedish

This chapter will describe a Swedish grammar and lexicon that can be used by PETFSG. The grammar is not a complete grammar for the Swedish language but a partial coverage of particle verbs and reflexives. Since a big part of the work described in this thesis has been to write a grammar that implements particle verbs it is necessary for such a grammar to deal with both verbs and reflexives. All verb particle constructions contains at least one verb. It is also the case that particle verb constructions can contain reflexives, hence the inclusion of reflexives in the grammar.

Verb particle constructions follow a schema (covered in more detail in section 4.3). The implemented grammar uses the suggested schema to implement the particle verbs. A particle verb construction is represented by letting the elements bound by valency be included in the COMPS list of the verb.

The complete lexicon and grammar is not included in the thesis, since they are quite big. All files however are available to download from the Internet. See Appendix A for information on how to obtain them.

The different FSs used will be shown as tables, generated from PETFSG, under the assumption that this will make it easier to read and understand them.

## 5.1 Noun inflection

Since Swedish sentences mostly requires a noun phrase to be complete it follows that a grammar for Swedish has to have some way to analyze them. From this it follows that the grammar has to cover the different forms nouns can take since nouns are parts of noun phrases. As shown in the schema for particle verb constructions (figure 3) it is often the case that objects can take part in them. In some constructions containing objects nouns appear in their definite form which means that the grammar needs some way to generate the definite form.

There are two cases in Swedish, nominative and genitive (Östen Dahl 1982). The implemented PETFSG grammar for Swedish does not handle the genitive form of nouns.

There are two genders in Swedish grammar, common and neuter (Östen Dahl 1982). Nouns that are preceded by the determiner *en* (*a/an*) are of the common gender. Nouns that are preceded by the determiner *ett* (*a/an*) are of the neuter gender. The definite form of a noun depends on both its gender and number. Nouns that describe non-persons are also assigned a gender. Nouns form their plural forms according to their declension. The following list shows the five different ways plural forms are created (Stroh-Wollin 1998).

- Some nouns form their plural form by adding the suffix **or**. For example *skola - skolor* (*school - schools*).
- For some nouns the suffix **ar** is added, as for the noun *biff - biffar* (*steak - steaks*).
- The suffix **(e)r** is also used for some nouns, consider *rot - rötter* (*root - roots*).
- Some nouns use the suffix *n* to form their plural form, *snöre - snören* (*string - strings*).
- Some nouns have the same form in both singular and plural form, as *hus - hus* (*house - houses*) or are totally altered *gås - gäss* (*goose - geese*).

The stem is altered for some nouns in their plural form (*rot - rötter*).

Nouns in the grammar is by default marked to form their plural form by adding the suffix **ar**. The plural form for nouns of other declensions can be formed by using the predicate `suppletive_form/3`.

In Swedish the definite form of nouns are formed according to the schema presented in table 4 (Stroh-Wollin 1998)

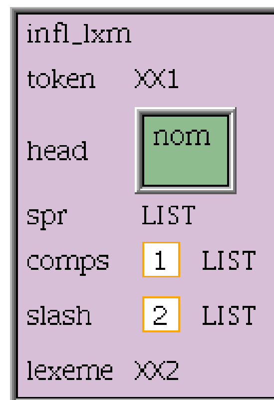
| singular indefinite | singular definite | plural indefinite | plural definite |
|---|---|---|---|
| flicka<br>*girl* | flickan<br>*the girl* | flickor<br>*girls* | flickorna<br>*the girls* |
| bok<br>*book* | boken<br>*the book* | böcker<br>*books* | böckerna<br>*the books* |
| snöre<br>*string* | snöret<br>*the string* | snören<br>*strings* | snörena<br>*the strings* |
| hus<br>*house* | huset<br>*the house* | hus<br>*houses* | husen<br>*the houses* |

Table 4: The definite forms of Swedish nouns

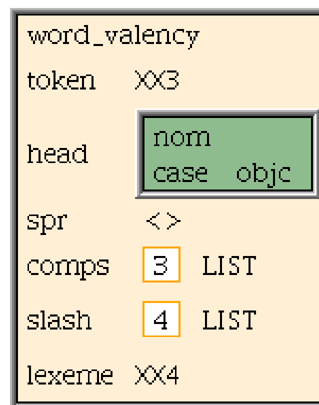
In the PETFSG grammar for Swedish, the lexical rule shown in figure 2 generates the definite forms of nouns.

Morph: morph_objekt

Input:




Output:




Sharing: XX1 = XX3 & XX2 = XX4 & 1 = 3 & 2 = 4

Figure 2: The lexical rule for definite forms

The following Prolog call asserts that, if nothing else is stated, all nouns should form their definite form by adding the suffix `et`.

```
infl_reg(morph_objekt,affx(et)).
```

It is however possible to override this behaviour for nouns whose definite form is different, according to the ways of forming the definite form of nouns presented earlier in this section.

## 5.2 Reflexives

Reflexive pronouns are used in sentences to corefer to the subject (Stroh-Wollin 1998).

In PETFSG reflexive pronouns have the following feature structure, as defined by the NFS `reflexive_pronoun`.

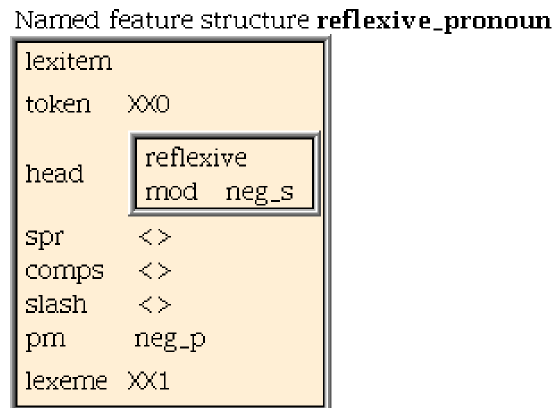



Figure 3: The NFS for reflexive pronouns

The reflexives must agree with the subject in a sentence, from which follows that lexical entries for reflexives must define agreement with the subject. Consider the lexical entry, shown in figure 4, for the reflexive *mig* (*myself*).

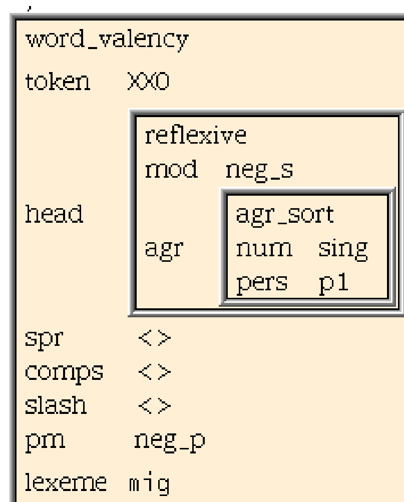



Figure 4: The FS for the reflexive pronoun *mig* (*myself*)

## 5.3 Verbs

In Swedish there is no agreement between a verb and its subject. The Swedish grammar handles infinitival verbs and the present and past tense of verbs. The lexical rule *v_infl_present* reshapes a lexeme to a FS that represents a verb form in its present tense. This includes both morphological and lexical adaptations. The lexical rule has the structure shown in figure 5.

```
Morph: v_present_tense

Input:
infl_lxm
token    XX1
head     [1] verb
spr      [2] LIST
comps    [3] LIST
slash    [4] LIST
lexeme   XX2

Output:
word_valency
token    XX3
head     [5] verb
             vform  present
             v_sf   [-,
                      -,
                      -]
spr      [6] LIST
comps    [7] LIST
slash    [8] LIST
lexeme   XX4

Sharing:  XX1 = XX3 & XX2 = XX4 & [1] = [5] & [2] = [6] & [3] = [7] & [4] = [8]
```

Figure 5: The lexical rule for present tense verbs

The lexical rule uses the NFS *present_form* in reshaping the verb to its new form, which gives the feature structure shown in figure 6.

Morphologically the surface form is generated by the morphological funtion *v_present_tense*, which is shown below.

```
infl_reg(v_present_tense,affx(er)).
```

This rule adds the suffix `er` to all verbs in their present tense. For verbs that do not form their present tense form this way it is possible to override the rule.

The infinitival forms of verbs are generated by the lexical rule *v_infl_infin*, which gives the feature structure shown in figure 7.

The morphological function connected to this lexical rule, shown below, lets the infinitival form of a verb be the same as the lexeme. It is possible to override this function.

```
infl_reg(v_infin_infl,id).
```

The past tense forms of verbs are generated by the lexical rule *v_infl_pret* shown in figure 8. The

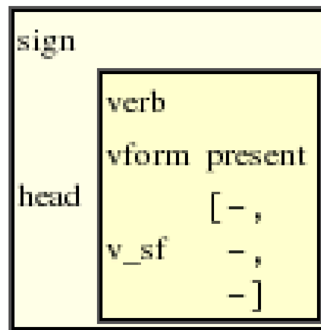



Figure 6: A feature structure that represents present tense verbs.

morphological rule *v_pret_infl*, shown below, alters the surface form of the verb form by adding the suffix *de* to the lexeme.

```
infl_reg(v_pret_infl,affx(de)).
```

It is possible to override this morphological rule for verbs that do not form their past tense form this way.

### 5.4 The structure of particle verb constructions

Given the structures for particle verb listed in section 4.3.1, I will in this chapter describe how they were implemented in the Swedish grammar for PETFSG.

The word order of verb particle constructions are illustrated in chapter 4.3.1. The listed data, in tables of word order forms for verb particles, show a rather clear and common structure for the different possible constructions of verb particles.

PETFSG is in no way limited to grammars written in a HPSG style. The Swedish grammar is written in a HPSG style because the English grammar as presented in Dahllöf (2003a) was written like that. The possible ways of adapting the grammar to handle particle verb constructions are limited to a number of strategies. One alternative is to list the particle as either a specifier or a complement to a lexical entry. This is done by simply listing the particle in one of the SPR or COMPS lists. Another possible way is to view the particle as a modifer to the verb it belongs to. By using this strategy the grammar rules handling modifiers probably has to be modified. A third strategy would be to view the particle as a lexical head and specify the verb to which the particle belongs to as a specifier to it. By using this strategy the particle would also be required to carry the information about the valency of the verbs, normally specified in the COMPS feature of the verb.

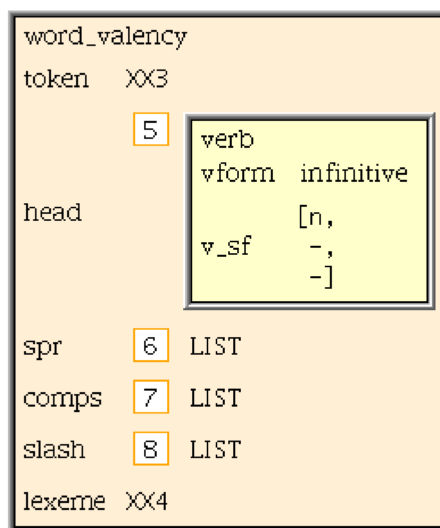
It is not an ideal strategy to view particles as modifiers to its verb since it does not really capture the different word order structures at hand. Modifiers are also not required elements and can furthermore be repeated any number of times whereas particles can only appear once. It is also the case that modifiers are possible to appear before the element they modify, although this fact is not reflected in the current grammar. This leaves the alternative of using the SPR and COMPS features. Since specifiers of a lexeme are supposed to appear before and not after the lexeme this is not possible to use, given the fact that particles appear after the verb to which they belong to. This means that the COMPS feature is the only feasible way of adding the capability of handling verb particles to the grammar. As for the strategy of making the particle a lexical head it is a grammatically dubious strategy and would also make the development of the lexicon harder.

By adding the different parts of the verb particle constructions to the COMPS list of each lexeme it is possible for the grammar to allow correct verb particle constructions and block incorrect ones. For each type of verb particle construction supported by the grammar a named feature structure is defined in

Lexical rule **v_infl_infin**
Morph: v_infin_infl

Input:

Output:




Sharing: XX1 = XX3 & XX2 = XX4 & 1 = 5 & 2 = 6 & 3 = 7 & 4 = 8

Figure 7: The lexical rule that generates infinitival verb forms.

which the needed complements for the verb particle construction are listed. When defining a lexeme all that is needed is to declare which form the particle should have. For an example, the lexeme of *hitta på* (*make up*), has the following structure in the lexicon:

```
hitta >>>
    [infl_lxm,
     nfs verb_particle_intransitive_adv,
     comps:{[head:[pform: <>up]], xNpObj},
     lexeme: <>hitta_på].
```

Note that since the NFS for the lexeme *hitta på* (*make up*) has a COMPS list consisting of two elements, the COMPS list of *hitta på* (*make up*) also has to consist of two elements. In this case the second element is a Prolog variable that will be instantiated in the parsing process.

Lexical rule **v_infl_pret**
Morph: v_pret_infl

Input:

infl_lxm
token XX1
head [1] verb
spr [2] LIST
comps [3] LIST
slash [4] LIST
lexeme XX2

Output:

word_valency
token XX3
head [5] verb
vform preterit
v_sf [+, -, -]
spr [6] LIST
comps [7] LIST
slash [8] LIST
lexeme XX4

Sharing: XX1 = XX3 & XX2 = XX4 & [1] = [5] & [2] = [6] & [3] = [7] & [4] = [8]

Figure 8: The lexical rule that generates past tense verb forms.

The NFS used to describe the various particles uses the feature `pform` to combine a verb with a particle.

This shows some regularity of the word order for the different types of verb particle constructions. HPSG, has three ways of defining word forms needed for a lexical item, either by defining them as possible modifiers to the lexeme or by listing them either as specifiers or complements. For particle verbs it is not possible to define the particles and other words as modifiers which leaves the use of specifiers and complements as the tool for adapting verb particle constructions. Since specifiers of a lexical item are to be placed before the lexical item as defined both in the theory of HPSG and in the grammar for PETFSG it is not possible to use them as a mean of defining verb particle constructions. This leaves the use of complements as the only feasible way to define them. This also fits very well with the word order form in which particle verbs appear.

## 5.5 Particles

As mentioned earlier particles can, although they appear to be of different parts of speech, be seen as adverbs. Verb particles are defined by the NFS shown in figure 9.

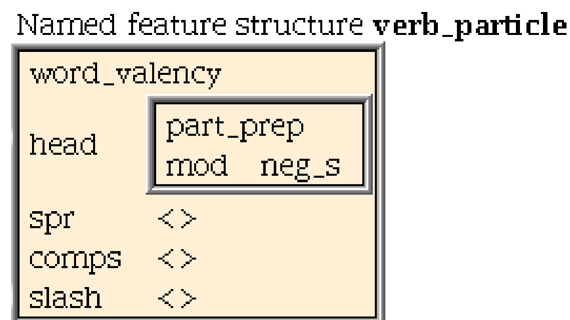



Figure 9: The NFS for verb particles

### 5.5.1 Verb and particle

This is implemented by the NFS shown in figure 10.

The lexical entry for *gå av* (*break*) looks as follows:

```
gå >>>
    [infl_lxm,
     nfs verb_particle_simple,
     comps: {[head:[pform: <>by]]},
     lexeme: <>gå_av].
```

### 5.5.2 Verb, adverbial and object

This is implemented by the NFS shown in figure 11

A lexical entry using this NFS is the following:

```
hämta >>>
    [infl_lxm,
     nfs verb_particle_obj,
     comps: {[head:[pform: <>ut]], xObj},
     lexeme: <>hämta_ut].
```

Named feature structure **verb_particle_simple**

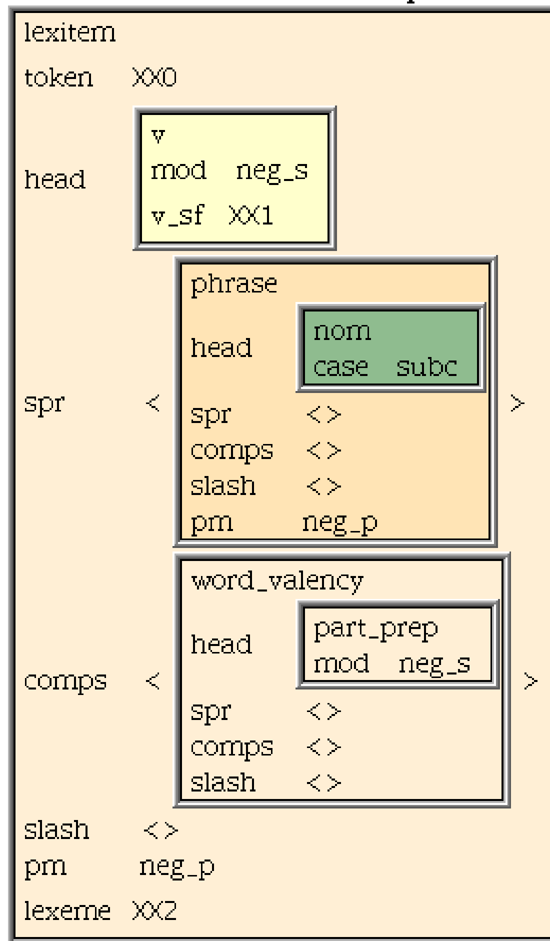



Figure 10: The NFS for simple particle verbs

### 5.5.3 Verb, particle, adverbial

Implemented by the NFS shown in figure 12.

A lexical entry using this NFS is the following:

```
hitta >>>
    [infl_lxm,
     nfs verb_particle_intransitive_adv,
     comps:{[head:[pform: <>på]], xNpObj},
     lexeme: <>hitta_på].
```

### 5.5.4 Verb, particle, object and adverbial

Implemented by the NFS shown in figure 13.

The follwing lexical entry uses this NFS:

```
sikta >>>
    [infl_lxm,
     nfs verb_part_obj_adv,
     comps: {[head:[pform: <>in]], xObj, xPrep},
     lexeme: <>sikta_in].
```

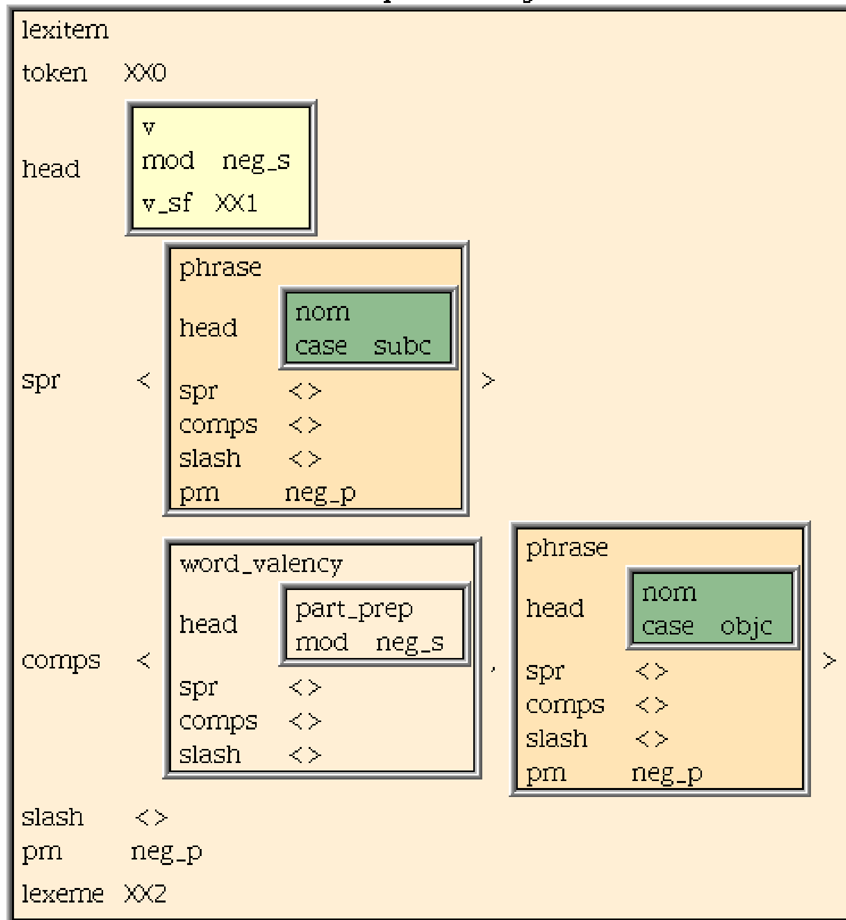



Figure 11: The NFS for particle verbs followed by an adverbial and an object

### 5.5.5 Verb, particle, indirect object and direct object

This is implemented by the NFS shown in figure 14.

A lexical entry using the NFS is the following:

```
ta >>>
    [infl_lxm,
     nfs verb_particle_ind_dir,
     comps: {[head:[pform: <>ifrån]], xObj, xObj},
     lexeme: <>ta_ifrån].
```

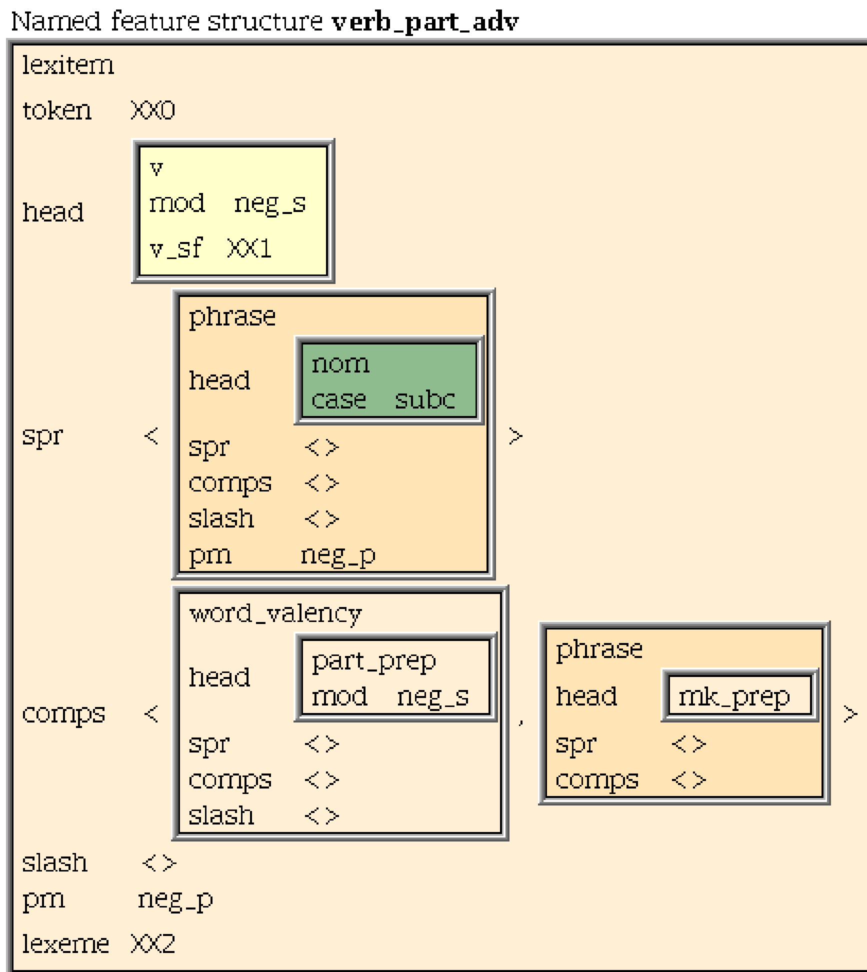



Figure 12: The NFS for a particle verb followed by its particle and an adverbial

# 6 Test suites

In this section I will give an overview of the test suite used to test the grammar. Apart from the sentences listed in table 3, the test suite consist of the following sentences:

- en pojke ler (*A boy smiles*)
- en pojke log (*A boy smiled*)
- två pojkar ler (*Two boys smile*)
- Mary avser att se en pojke (*Mary intends to see a boy*)
- Mary avser att läsa en bok (*Mary intends to read a book*)
- två pojkar åt tre äpplen (*Two boys ate three apples*)
- en pojke äter ett äpple (*A boy eats an apple*)
- en flicka raderar två filer (*A girl erases two files*)

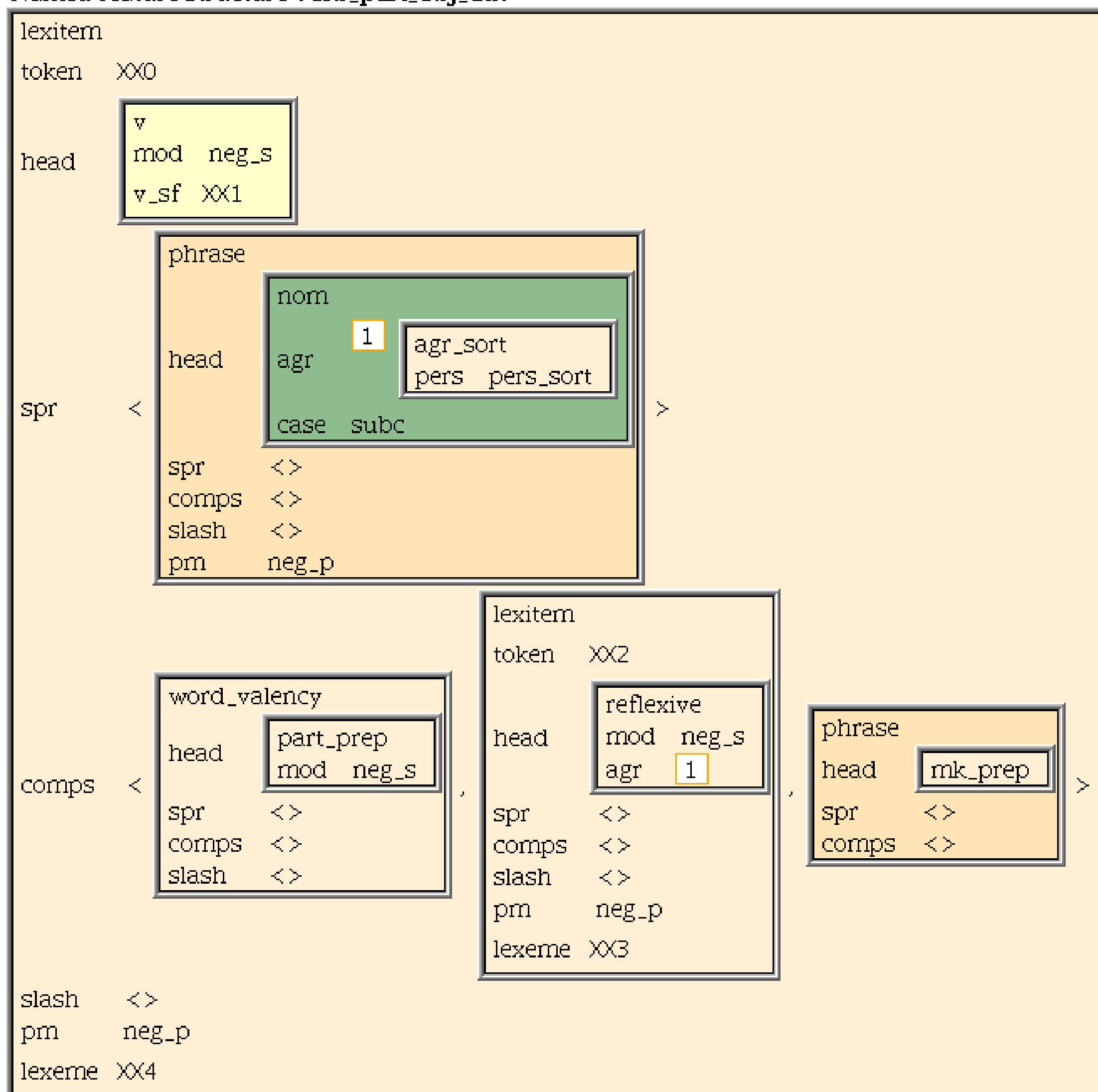



Figure 13: The NFS for a particle verb followed by its particle, an object and an adverbial

## 6.1 Summary of the test suite

The sentences in the test suite illustrates that the implemented grammar can handle the different particle verb constructions that are grammatically correct in Swedish.

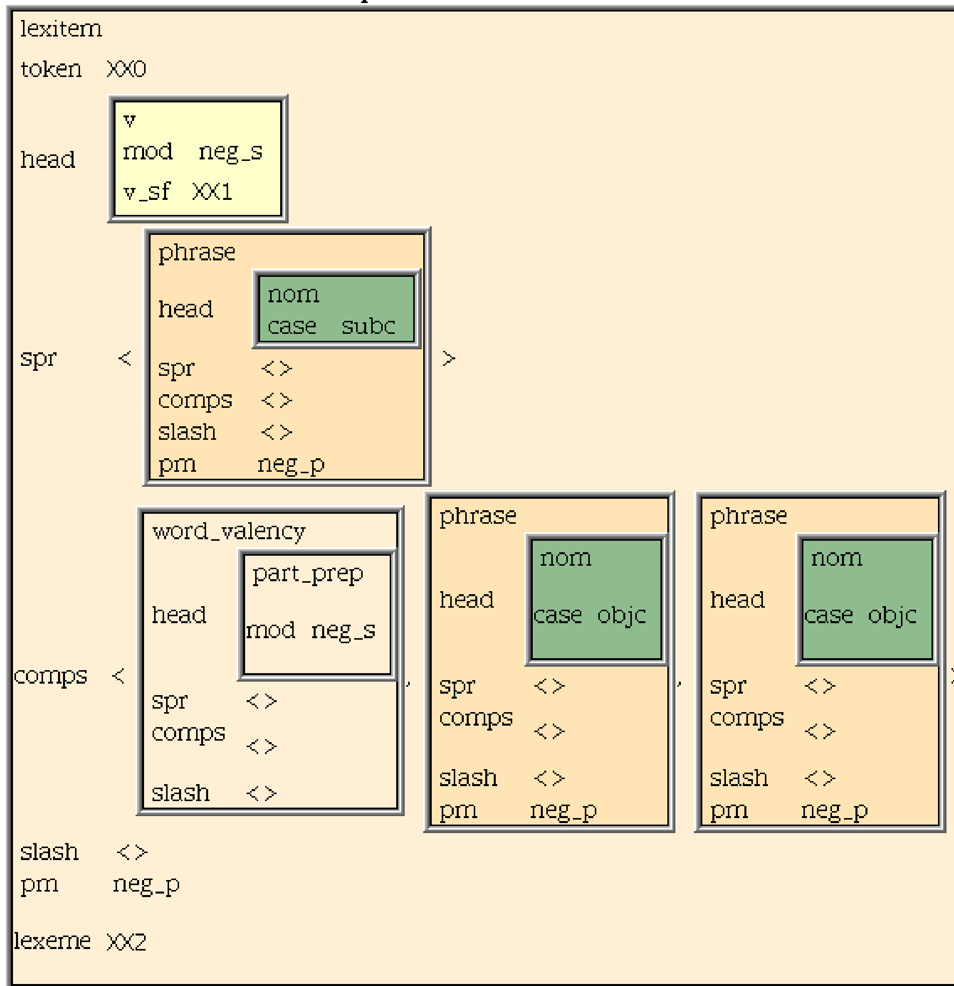



Figure 14: The NFS for particle verbs followed by an indirect object and a direct object

# 7 A stand alone GUI for PETFSG

Previously there has been two ways of interacting with PETFSG, by using the grammar directly from Prolog or via the HTML/CGI interface. Both ways can be cumbersome for users, interacting directly with the parser through Prolog is time consuming and error prone due to the amount of typing needed and the output of the interactions with the grammar can be hard to interpret. The HTML/CGI interface solves both these problems but adds others. Since the HTML/CGI version needs to run via a web server each user needs to have a CGI-directory defined for the web server, which can be unpractical and the requirement of a web server is really quite unnecessary for an application of this kind. Furthermore, it is only possible to view one element of grammar and lexicon at the time, which can make it both hard and time consuming to develop or even understand a grammar.

In order to solve these problems and making the PETFSG easier to use there was a need to implement a stand alone graphical user interface (GUI) application. This application should have the following features:

- No web server needed.
- The GUI should be easy to use.
- The application should be easy to set up.

- The application should use saved state files created by PETFSG.

The application, called PetFsgGui, is written completely in Java and uses Java classes provided by SICS[1] to interact with Prolog. These classes are distributed in a package called `Jasper`. Although PETFSG works both with SICStus and SWI-Prolog[2] PetFsgGui only uses SICStus at the moment. According to the documentation for the Java bindings of SWI-Prolog it does at the time not support the use of Prolog programs written in modules. This fact makes it impossible to use SWI-Prolog together with PetFsgGui to interact with PETFSG at the time being. However, it is most likely that the support for modular Prolog programs will be incorporated some time in the near future.

### 7.1 Overview of the programme

This section will give a description of how PetFsgGui works, for more detailed information on how to compile the application, a listing of all methods and classes in the program, see Appendix B.

The program starts a Prolog engine which it communicates later with to execute the actions that the user generates from the GUI. Output generated by PETFSG is written to a file that PetFsgGui reads and presents to the user. The format of the output is HTML and is presented as interpreted HTML to the user, which results in an easily read format. The generated output file can in some cases be quite large and depending on the computer environment in which the program is run, it can take some time to read it.

As input the program takes a saved state file containing a PETFSG application (grammar, lexicon and parser). This file is generated from PETFSG with the aid of a Prolog program, see Dahllöf (2003b) for information on how this is done.

PetFsgGui consists of the following classes:

- `PetFsgGui`
  The main graphical user interface.

- `Parser`
  Manages all Prolog-calls.

- `PrologException`
  A general exception class used in interaction with Prolog.

- `Detail`
  A graphical user interface component that shows details about chosen things in the interface.

The classes listed above are described in the following chapters. For a more detailed account of them, see Appendix B.

Figure 15 shows the structure of the classes, Prolog and the saved state file and tries to illustrate how the different parts of the program interact.

### 7.2 The graphical user interface

Figure 16 shows a screen shot of the main window of PetFsgGui, without a grammar loaded. This is the first thing that is shown to the user when the GUI is loaded.

The grammars that the application can use are supposed to be in the form of saved state files created by the Prolog predicate *save_grammar_state/0*. The compilation is easily made with the Prolog program `make_tds.pl` which is available together with the rest of the source code that makes PETFSG. Saved state files contain compiled Prolog code. The saved state file must be created using SICStus Prolog. No saved state file is loaded by default when the program is started. To load a saved state file the alternative `load grammar` in the menu `options` is used which will present the user with a dialog box from

[1] See: http://www.sics.se

[2] SWI-Prolog is a free Prolog implementation that can be downloaded from *http://www.swi-prolog.org*

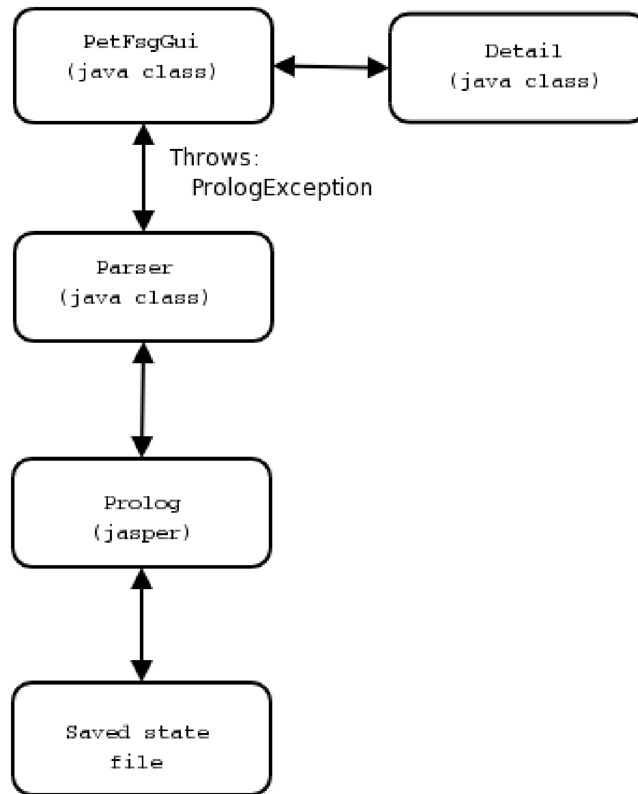



Figure 15: The structure of the program

Figure 16: The main window with no grammar loaded

which the user can choose a saved state file to load. After a saved state file is loaded sentences can be

parsed by typing them in the text field to the right of the label `sentence` followed by pressing either the `return` key (on the keyboard) or by pressing the button `parse`. The result is shown in the white colored area in the application.

In many cases the output from PETFSG contains hyperlinks. These hyperlinks are possible to use in PetFsgGui and depending upon what the hyperlinks actually refers to the information will be presented in the main window of PetFsgGui. Hyperlinks can for example refer to NFSs or lexemes of the grammar.

Any predefined actions available for the grammar are listed in the drop-down combo named `show`, by choosing an item in the list the output of the action is shown in the main window. The predefined actions can be of the following types:

- Showing all defined NFS's.
- Showing the type hierarchy for the grammar.
- Showing all defined lexical rules.
- Showing all defined lexemes.
- Showing a list of all generated lexical forms.
- Showing all rules of the grammar.
- Showing general information about the PETFSG application.

Note that the set of predefined actions can differ depending on which version of PETFSG is used.

In order to make the testing of grammars and lexicons easier it is possible to define sets of test sentences in separate files. These sets of sentences are called suites and have to be declared in a certain structure, see Dahllöf (2003b) for a detailed account on how this is done. Suites that are defined for a grammar that is loaded are listed in the drop down combo called `suites`. By choosing a suite from the list its sentences will be parsed and the output will be presented in the white colored area. Note that suites of test sentences can not be defined in PetFsgGui itself but has to be defined when compiling the grammar.

The parser that PETFSG uses is a chart parser and it is possible to view the chart created during parsing, this option can be switched on or off by clicking on the buttons `yes` respectively `no` to the right of the label `show chart`. The chart table created during parsing will be shown in the main window.

Grammars in PETFSG can have a number of predefined post parse procedures implemented (Dahllöf 2003b). If any are available they are listed in the drop down combo called `post parse procedures`. By selecting one from the list it will be applied next time a sentence is parsed. The available postparse procedures are retrieved from PETFSG. Note that this list can differ depending on which version of PETFSG is used.

For detailed information on methods in the classes see Appendix B.

Some changes were made to the PETFSG code base in order to make the implementation of PetFsgGui. The changes are all of the kind that they add functionality to return information from PETFSG in a format that is easier to read in Java. These changes do not alter the formalism in any way but it is necessary for saved state files to be compiled using this updated version of PETFSG for them to be possible to use in PetFsgGui.

It is possible to view detailed information about entities of both grammar and lexicon by clicking on the hyperlinks presented in the GUI. Detailed information is shown in a separate window, as the window shown in Figure 17. It is not necessary to close other detail windows when opening another. This makes the testing and development of grammars much easier.

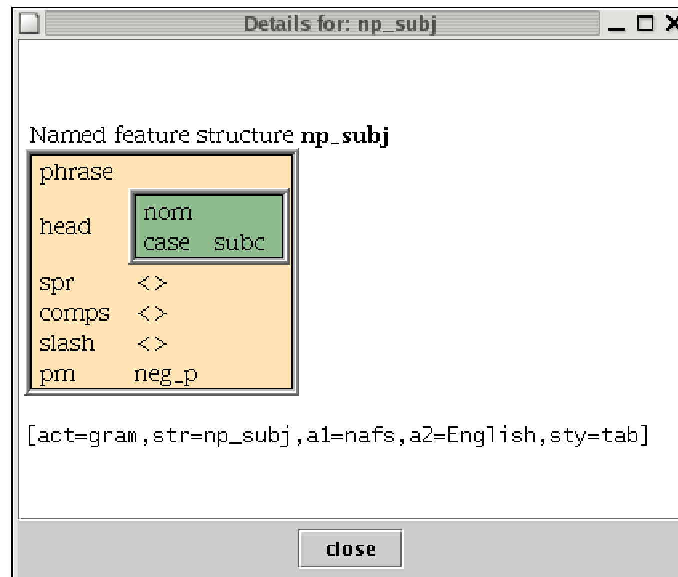



Figure 17: Details of a NFS

### 7.3 The graphical user interface (`PetFsgGui`)

The GUI is written using the Java Foundation Classes (JFC, often called Swing). The class does not communicate directly with Prolog in any way but merely manages the way that information from PETFSG is presented to the user as well as it interprets the actions that the user performs and directs them to other classes, such as `Parser` which communicates directly with Prolog, and PETFSG.

The class lets the user choose a saved state file to load from a graphical dialog.

It is also possible for the user to save the output from PETFSG to a file for future reference. This file contains HTML and should be possible to view in any program, such as a web browser, that can display HTML.

### 7.4 Interaction with Prolog (`Parser`)

Interaction with Prolog is handled with the aid of the Java classes provided in the package *Jasper* which contains classes that makes it possible to run a Prolog environment embedded in a Java object. These classes are provided by SICS. In order to make the GUI application more structured and easy to develop all interaction with Jasper is performed in the class `Parser`. Apart from merely making the program structured it also makes it easier to adopt the GUI to other Prolog environments (most likely *SWI-Prolog*) when so desired.

The class `Parser` contains methods for restoring saved state files as well as for executing queries in the Prolog-environment. For a detailed list of what methods are available, their arguments and return types, in the class *Parser* see Appendix B.

### 7.5 Details on chosen grammar elements (`Detail`)

This class shows information about a grammar item chosen by the user. The information is shown in a separate window. It is possible to have any number of detail windows open at the same time, as long as the computer's memory is sufficient. The closing of one detail window does not affect other detail windows that happens to be open. Apart from being able to show a HTML file it does nothing. As the class `PetFsgGui` it is written using JFC.

### 7.6 Exceptions

In order to make the error handling uniform, the program uses the class `PrologException` for handling errors and exceptions that may occur in communicating with Prolog. The class package *Jasper* provides a number of exceptions but these are obviously specific for SICStus. By having a uniform way of dealing with exceptions and errors it is most likely easier to adopt PetFsgGui to other Prolog engines in the future.

This class inherits the standard class *Exception* with no additions and contains a string, defined in the different classes, that explains the nature of the error or exception.

### 7.7 Summary of the graphical user interface

The implemented application, PetFsgGui, provides a fast and stable environment that makes the use of PETFSG easier. The possibility to have several windows showing detailed information about the different elements of both grammar and lexicon makes the development of grammars and lexicons both easier and faster.

Although the application is written completely in Java it is quite fast and responsive. Java is generally not known as a fast environment in which to execute GUI applications in and generally considered to be slow and resource consuming. The most time consuming task of the PetFsg is to read the output file generated by PETFSG. This is due to the fact that for some operations these can be quite big.

At the time PetFsgGui only supports the SICStus variant of PETFSG. The version of SICStus used to develop the application is rather old and in later versions of SICStus the `Jasper` classes have been changed. This means that parts of PetFsgGui must be rewritten if to be used together with later versions of SICStus. It is unclear whether the `Jasper` classes will ever be finalized in that they will not change from version to version of SICStus.

As for SWI-Prolog, the other Prolog environment that PETFSG supports, at the time it does exist a Java interface that makes it possible to be used from Java[3]. However, unfortunately this interface does not support Prolog programs that are written in modules. This is most likely to change in the future which will make it possible to extend PetFsg to use it as well.

All code of PetFsgGui will be released according to the license of GNU General Public License[4] which hopefully will result in that the program will be adapted and subjected to more development even if it's author lacks the time and energy to do it himself.

PetFsgGui has been used by Mats Dahllöf and a group of his students in a course on syntactic theory. The overall impression of PetFsgGui has been positive and according to the students it has been easy to use and made the use of PETFSG even more pleasant than before.

## 8 Conclusions

The aim of the work described in this thesis has been twofolded. One aim was to implement a Swedish grammar for PETFSG that implements some aspects of particle verbs in Swedish. The other aim of the work has been to construct a graphical user interface for PETFSG that would make PETFSG easier to use.

The implemented grammars adhers to the data presented in the background sections. Furthermore it is implemented in a way that avoids dubious grammatical solutions, that is the grammar fits well in with both the presented grammatical data and the grammatical framework that PETFSG provides.

Since the implemented grammar does not totally cover Swedish, there is much room for further development. An interesting grammatical construction to implement could be passive constructions which are not handled in the current grammar.

[3]The interface and information about it can be found at *http://sourceforge.net/projects/jpl/*

[4]See *http://www.fsf.org* for more information

Even though the implemented grammar is rather complete compared to that presented in Sag et al. (2003), there are some things missing. The coordination rule, as presented in Sag et al. (2003) is not implemented in the grammar described in this thesis. An implementation of this rule would allow constructions not handled by this grammar.

PetFsgGui, the GUI for PETFSG, makes the use of PETFSG simpler and the need of a web server redundant. As of now there are no known bugs in the programme. There is however room for further development and improvement of the programme. A very useful extension of its' capabilities would be to make it possible to edit both grammar and lexicon files from within the programme. The application would also benefit from having the capability to compile saved state files.

## References

Copestake, A., Flickinger, D. and Sag, I. A. (1999). Minimal recursion semantics an introduction, *Technical report*, CSLI, Stanford University.

Dahllöf, M. (2003a). An implementation of token dependency semantics, ruul nr 36, *Technical report*, Department of Linguistics, Uppsala University.

Dahllöf, M. (2003b). Prolog-embedding typed feature structure grammar (petfsg-ii.2) and grammar tool, ruul nr 36, *Technical report*, Department of Linguistics, Uppsala University.

Holm, L. and Larsson, K. (1976). *Svenska meningar*, Studentlitteratur.

Sag, I. A., Wasow, T. and Bender, E. M. (2003). *Syntactic Theory A Formal Introduction*, CSLI Publications.

Stroh-Wollin, U. (1998). *Koncentrerad nusvensk formlära och syntax*, Studentlitteratur.

Teleman, U. (1974). *Manual för grammatisk beskrivning av talad och skriven svenska*, Lundastudier i nordisk språkvetenskap, serie C nr 6, 2:a tryckningen, Lund.

Teleman, U., Hellberg, S. and Andersson, E. (1999). *Svenska akademiens grammatik*, Norstedts.

Thorell, O. (1977). *Svensk grammatik*, Scandinavian University Books. Second edition.

Östen Dahl (1982). *Grammatik*, Studentlitteratur.

# A Grammar and lexicon

## A.1 Grammar, lexicon and test suites

In this section the grammar and lexicon will be listed. For instructions on how to compile an PETFSG application see Dahllöf (2003b).

## A.2 The grammar

The grammar that this thesis covers will, for a foreseeable future, be available to download from the following URL:

`http://stp.ling.uu.se/~tomasen/petfsg/gram_tds.pl`

## A.3 The lexicon

The lexicon that this thesis covers will, for a foreseeable future, be available to download from the following URL:

`http://stp.ling.uu.se/~tomasen/petfsg/lex_tds.pl`

## A.4 Test-suites

The test suite is stored in a file named `suites_tds.pl`, is available to download from the following URL:

`http://stp.ling.uu.se/~tomasen/petfsg/suites_tds.pl`

The test suite will, for a foreseeable future, be possible to download from the following URL:

`http://stp.ling.uu.se/~tomasen/petfsg`

# B The GUI for PETFSG

## B.1 Source code

The source code for PetFsgGui is available to download from the following URL:

`http://stp.ling.uu.se/~tomasen/petfsg/petfsggui/`

The application consists of the following files:

- `PetFsgGui.java`
- `Parser.java`
- `Detail.java`
- `PrologException.java`

Note that all listed files are neccessary to download in order to compile the application.

## B.2 The structure of the program

Here I will describe the structure of the program and its classes. The program is developed and intended to run on computers using the GNU/Linux operating system, it has not been tested on any other platform. It should in principle work on any operating system as long as the required versions of both SICStus Prolog and Java are installed.

## B.3 Required software

The following software are necessary for running the program.

- **Java**
  For merely running the program it is only necessary to have a *Java Runtime Environment* (*JRE*) installed. In order to compile the program a *Java Development Kit* (*JDK*) is needed.
  The program was created using Sun's *jdk1.4.1_01*[5].

- **SICStus Prolog**
  Although the PETFSG-application can use SWI Prolog this application only uses SICStus Prolog. The version used is 3.8.3. SICStus provides Java classes that are needed for this application, thus it is required that these are installed as well.
  In later versions of SICStus Prolog the Java interface is changed and the current code may or may not work as intended.

- **Redhat GNU/Linux 9**
  The application was developed and tested using this operating system[6].

## B.4 Required hardware

The program is developed and tested on a computer with the following equipment:

- AMD Duron 750 MHz
- 256 MB RAM

The program should work on any computer that supports the required software.

## B.5 Prerequisites for compiling the application

The interaction with SICStus Prolog is made possible with the classes available in the package `Jasper`, these must be available for the Java compiler. The easiest way to accomplish this is to set the environment variable `CLASSPATH` to point to the right directories. Use the following command for this:

```
export CLASSPATH=.:/local/lib/sicstus-3.8.3/bin:/local/lib/sicstus-3.8.3
```

It is also necessary to point out where the libraries for threaded Java applications are found[7], this is done by setting the environment variable `LD_LIBRARY_PATH` to point to the right directory. Use the following command for this[8]:

```
export LD_LIBRARY_PATH=/usr/java/j2sdk1.4.1_01/jre/lib/i386/native_threads/
```

The steps above can be made permanent by adding the lines to one, or both, of the files[9]:

- `bashrc`. Commands in this file will be executed any time a shell is started.
- `bash_profile`. Command in this file will be executed at login.

Note that these instructions only hold true for Unix-like operating systems.

[5] See *http://java.sun.com* for downloads and documentation

[6] For more information see: *http://www.redhat.com*

[7] Depending on the installation this may very well already be done

[8] This holds for the environment in which the program was developed. If not read the documentation for Java.

[9] The following holds for the shell **bash** (textitbourne again shell), distributed by the Free Software Foundation (*http://www.fsf.org*).Read the manual for more information or see: *http://www.fsf.org*

## B.6 Recommended structure for installation

At the moment this is totally up to the user although it is really recommended that all the source code is placed in a directory dedicated only to this application.

## B.7 Compiling the program

There is a `Makefile` provided with the source code that automatically compiles the source code to byte code. The `Makefile` also automatically sets all environment variables that are needed for compiling the program.

To compile the program using this `Makefile` do the following[10]:

1. Start a terminal, such as `xterm`, and move to the directory containing the source code.
2. Type the following:

   ```
   # make all
   ```

   Note that the sign # is to be interpreted as the prompt sign.
3. If no error messages are displayed the application is compiled and ready to use.

### B.7.1 Makefile

The contents of the *Makefile* is listed below:

```
## Compiles PetFsgGui                                  ##
## Written by: Tomas Englund (tomasen@stp.ling.uu.se) ##

ld = /usr/java/j2sdk1.4.1_01/jre/lib/i386/native_threads/

classpath = .:/local/lib/sicstus-3.8.3/bin:/local/lib/sicstus-3.8.3

all : PetFsgGui.class Parser.class Detail.class PrologException.class

clean :
        echo Removes *all* class-files!
        rm *.class
        rm *~
PetFsgGui.class : PetFsgGui.java
        export LD_LIBRARY_PATH=\$(ld)
        javac -classpath \$(classpath) PetFsgGui.java

Parser.class : Parser.java
        export LD_LIBRARY_PATH=\$(ld)
        javac -classpath \$(classpath) Parser.java

Detail.class : Detail.java
        export LD_LIBRARY_PATH=\$(ld)
        javac -classpath \$(classpath) Detail.java
```

[10] Note that this requires that the program GNU `make` is installed. For more information about *GNU make* see: *http://www.fsf.org/directory/devel/build/make.html*

```
PrologException.class : PrologException.java
        export LD_LIBRARY_PATH=\$(ld)
        javac -classpath \$(classpath) PrologException.java

Finish.class : Finish.java
        export LD_LIBRARY_PATH=\$(ld)
```

## B.8 Running the program

In order to make the handling of the program easier I have written a small shell script which exports all necessary environment variables and starts the program. This shell script can be found in Appendix B.8.1.

The environment variables that need to be set are:

1. `LD_LIBRARY_PATH`
   This variable should point to the library containing the Java libraries for threaded applications. In this case it points to the directory */usr/java/j2sdk1.4.1_01/jre/lib/i386/native_threads/*

2. `CLASSPATH`
   This variable should point to the directory containing the classes in the *Jasper* package as well as the directory containing the *jar* file *jasper.jar*. In this case it points to the following directories: */local/lib/sicstus-3.8.3/bin* and */local/lib/sicstus-3.8.3*

### B.8.1 The script that runs the program

In order to make the use of the application easy for the user there is a small shell script, `runGui.sh`, that sets all necessary environment variables to their correct values and starts the application. This script is listed below:

```
#! /bin/bash

echo exporting LD_LIBRARY_PATH
export LD_LIBRARY_PATH=/usr/java/j2sdk1.4.1_01/jre/lib/i386\
/native_threads/:/usr/java/j2sdk1.4.1_01/jre/lib/i386/native_threads/

echo exporting CLASSPATH
export CLASSPATH=.:/local/lib/sicstus-3.8.3/bin:/local/lib/sicstus-3.8.3

echo ...starting application!
java -Djava.library.path=/local/lib/ -Dsicstus.path=/local/lib/sicstus-3.8.3/ \
-classpath .:/loca\l/lib/sicstus-3.8.3/bin/jasper.jar PetFsgGui
```

Note that this script is written for Unix-like operating systems.

## B.9 A typical call to Prolog

The documentation provided for `Jasper` by SICS is not too descriptive and quite poorly organized. Therefore an example of a call to Prolog is shown and described below, this code is taken from the class `Parser` but adopted to suite its descriptive purpose.

The following code is taken from the method `getPostParse()` which returns an array of `String` objects which represents all defined post parse procedures. Note that all code apart from the declaration of the variables must be enclosed within a `try/catch`-block as the operations can throw exceptions.

The code calls the predicate `postparse_procedure/1`, in Prolog the call would look like this:

```
main_petfsg:postparse_procedures(L).
```

where `L` is a variable which hopefully will be instantiated to a list containing the names of all post parse procedures.

Step by step instruction:

1. Define all necessary variables.

   ```
   SPPredicate p;
   SPQuery q;
   SPTerm result;
   String answer = "";
   ```

   This does not need to be done within a `try/catch`-block.

2. Create the terms needed for the call and make it a variable

   ```
   result = new SPTerm(s,"Postparse");
   result.putVariable();
   ```

3. Create an object which represents the predicate we wish to call (`postparse_procedures`)

   ```
   p = new SPPredicate(s,"postparse_procedures",1,"main_petfsg");
   ```

   where:

   - **`s`** refers to which SICStus object the predicate should be passed.
   - **`postparse_procedures`** is the name of the predicate we want to execute.
   - **`1`** is the arity of the predicate.
   - **`main_petfsg`** is the module in which the predicate can be found.

4. Try to open a query to Prolog.

   ```
   q = s.openQuery(p,new SPTerm[] {result});
   ```

   Where:

   - **`p`** is the predicate we want to execute.
   - **`new SPTerm[] {result}`** is the list of arguments for the call (note that these must be passed as an array here).

5. Try to execute the call to the predicate.

   ```
   q.nextSolution();
   ```

   Normally, and as is the case in the code, it is a good idea to put this call in a `if`-statement which makes it possible to check whether the call was successful or not.

6. Close the question.

   ```
   q.close();
   ```

   This is quite important as it presumably releases allocated resources back to the operating system.

7. Convert the variable to a `String`-object.

   ```
   answer = result.toString();
   ```

8. Continue with what needs to be done.

## B.10 Java classes in the application

The application consists of the following files/classes[11]:

- **`PetFsgGui`**
  This class contains the graphical user interface.
- **`Parser`**
  This class handle all calls to Prolog.
- **`PrologException`**
  An exception used in the application.
- **`Detail`**
  Shows details about elements of the grammar.

## B.11 Class: PetfsgGui

This class is the actual graphical user interface. The class contains the following methods and classes:
**Methods:**

- **`PetFsgGui()`**
  The constructor for the class. Creates all graphical components.

  **Arguments:**

  - None.

  **Returns:**

  - Nothing.
    **Type:** `void`

- **`setDebug()`**
  Makes the program either to show or not to show debug information.

  **Arguments:**

  - The value for debug.
    **Type:** `boolean`

- **`addSentence()`**
  Adds the sentence the user last entered to a visible so that it can be reused.

  **Arguments:**

  - None.

  **Returns:**

  - Nothing.
    **Type:** `void`

- **`parseSentence()`**
  Tries to parse the sentence the user has typed in or chosen from the list of available sentences.

  **Arguments:**

[11] The suffix *java* is dropped in the list below

– None.

**Returns:**

– Nothing.
**Type:** `void`

- **`showSelected()`**
  Executes one of the predefined tasks that the user has chosen.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`chooseAndLoadFile()`**
  Shows a file chooser dialog in which the user can choose a saved state file to load. The name of the chosen file is stored in the global variable `filename`.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`restorePrologFile()`**
  Tries to restore the saved state file the user has chosen.

  **Arguments:**

  – None

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`setGrammarInfo()`**
  Collects information about the grammar and shows it in the graphical user interface.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`enableInput()`**
  Disables all graphical objects in the interface.

  **Arguments:**

– None.

**Returns:**

– Nothing.
**Type:** `void`

- **`disableInput()`**
  Enables all graphical objects in the interface.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`showSuite()`**
  Runs the selected suite of sentences.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`closeWindow()`**
  Currently not used. Tries to clean up everything before the application is closed.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.
  **Type:** `void`

- **`saveOutput()`**
  Saves the HTML output shown in the window to a file. The name of the file must end with the suffix `.html`.

  **Arguments:**

  – None.

  **Returns:**

  – Nothing.

- **`showAbout()`**
  Shows information about the application.

  **Arguments:**

  – None.

**Returns:**

- Nothing.

- **`main()`**
  Starts the application and creates the graphical user interface.

  **Arguments:**

  - A list of arguments.
    **Type:** `Array of String`

  **Returns:**

  - Nothing.
    **Type:** `void`

**Inner classes:**

- **`indataListener`**
  A listener for the various drop down combo menus in the interface.

- **`menuListener`**
  A listener for the menu and its items.

- **`wordListener`**
  A listener for the HTML that are shown in the main application window.

- **`window`**
  A listener for the window and some of the events that it raises. Currently not used.

### B.12 Class: Parser

This class handles all interaction with SICStus Prolog. This is made possible by using the package *Jasper*. For information on *Jasper*, see the relevant web pages at *http://www.sics.se*.

The class contains the following methods:

- **`Parser()`**
  Constructor for the class, tries to create a new SICStus object which is a Prolog interpretor.
  **Arguments:**

  - none

  **Returns:**

  - Nothing.
    **Type:** `void`

- **`setDebug()`**
  Sets the debug flag to `true` which results in that debug information is written to standard output when the application is run.
  It is recommended that this is set to `false` because of all information that is written to *standard out*.
  **Arguments:**

  - A boolean value that shows whether debug information should be shown or not.
    **Type:** `boolean`

**Returns:**

- Nothing.
  **Type:** `void`

- **`restoreParser()`**
  Tries to restore a parser from file. The file containing the grammar should be a saved state file. For information on how to do this see (Dahllöf 2003b) where it is described in detail.
  **Arguments:**
  - The name of the grammar file that should be restored.
    **Type:** `String`

  **Returns:**
  - Nothing.
    **Type:** `void`

- **`runSuite()`**
  Runs a predefined suite. These are defined when creating the grammar.
  **Arguments:**
  - The name of the suite that should be run as a string. Note that the suites are defined in the grammar.
    **Type:** `String`

  **Returns:**
  - The output from the grammar.
    **Type:** `String`

- **`changeOutput()`**
  Sets the output style to one of the following:
  - **`tab`**
    The output is written as HTML.
  - **`text`**
    The output is written as plain ASCII.

  **Arguments:**
  - The type of the output, that is one of the previous.
    **Type:** `String`

  **Returns:**
  - The output from the grammar.
    **Type:** `String`

- **`getWordInfo()`**
  Returns the information the grammar has about a selected word.
  **Arguments:**
  - The word that the user wants information about.
    **Type:** `String`

  **Returns:**

- The output from the grammar.
  **Type:** `String`

- **`getGramInfo()`**
  Returns information about something of the following:

  - A named feature structures.
  - A lexical rule.
  - A grammatical rule.

  **Arguments:**

  - The action that the grammar should perform.
    **Type:** `String`
  - The item the information should be written about.
    **Type:** `String`
  - The value of the parameter **`a1`**
    **Type:** `String`
  - The value of the parameter **`a2`**
    **Type:** `String`

  **Returns:**

  - The output from the grammar.
    **Type:** `String`

- **`getSuites()`**
  Returns an array of all predefined suites of test sentences that are defined for the grammar.

  **Arguments:**

  - None
    **Type:** `void`

  **Returns:**

  - An array of strings (the defined suites).
    **Type:** `An array of String`

- **`getVersion()`**
  Returns information about when the grammar was compiled and what version of PETFSG is used.

  **Arguments:**

  - None

  **Returns:**

  - The output from the grammar
    **Type:** `String`

- **`getCompiledTime()`**
  Returns a string containing the date and time when the grammar was compiled.

  **Arguments:**

  - None.

**Returns:**

- The output from the grammar.
  **Type:** `String`

- **`getTimeFromParser()`**
  Returns the current time from parser.

  **Arguments:**

  - None.

  **Returns:**

  - The output from the grammar.
    **Type:** `String`

- **`closeParser()`**
  Tries to close the Prolog interpreter. It is strongly advised that this method is not used because it causes the Java engine to crash. This is most likely due to bugs in the `Jasper` package.

  **Arguments:**

  - None.

  **Returns:**

  - Nothing.
    **Type:** `void`

- **`getOutput()`**
  Returns the output of the Prolog statement last executed. The output is stored in a file named `out.html` in the directory where the program was started from.

  **Arguments:**

  - None.

  **Returns:**

  - The output from the grammar.
    **Type:** `String`

- **`getPostParse()`**
  Returns an array containing all post parse procedures that are available for the grammar.

  **Arguments:**

  - None.

  **Returns:**

  - The available post parse procedures as an array of string
    **Type:** `An array of String`

- **`parseSentence()`**
  Parses a sentence and returns the output as a string.

  **Arguments:**

- The sentence to parse.
  **Type:** `An array of String`
- The value of the option **`a1`**.
  **Type:** `String`
- The value of the option **`a2`**.
  **Type:** `String`

**Returns:**

- The output from the grammar.
  **Type:** `String`

- **`getEdgeInfo()`**
  Returns information about a specified edge in a chart. This can be an active or inactive edge.

  **Arguments:**

  - The action to be performed by the grammar.
    **Type:** `String`
  - The sentence to parse.
    **Type:** `String`
  - The value for the option **`a1`**.
    **Type:** `String`
  - The value for the option **`a2`**.
    **Type:** `String`

  **Returns:**

  - The output from the grammar.
    **Type:** `String`

- **`getPredefined()`**
  Returns the output of the standard questions to the grammar. The questions that can be executed are the following:

  - **`nafs`**
    Returns all named feature structures.
  - **`types`**
    Returns the type hierarchy for the grammar.
  - **`lexrules`**
    Returns all lexical rules in the grammar.
  - **`lilex`**
    Returns all lexemes in the grammar.
  - **`lifor`**
    Returns all lexical forms in the grammar.
  - **`rules`**
    Returns all rules for the grammar.
  - **`gener`**
    Returns general information about the grammar.

  **Arguments:**

- The act to perform, one of the previous.
  **Type:** `String`

**Returns:**

- The output from the grammar.
  **Type:** `String`

- **`getNfsFromFile()`**
  Returns a string containing all named feature structures that are available for the grammar. This method calls the method `getPredefined()` with the argument `nafs`.

  **Arguments:**

  - None.

  **Returns:**

  - The output from the grammar as a string.
    **Type:** `String`

- **`getTypesFromFile()`**
  Returns a string containing all types that are available in the grammar. This method calls the method `getPredefined()` with the argument `types`.

  **Arguments:**

  - None.

  **Returns:**

  - A string containing all types
    **Type:** `String`

- **`getGeneralInfo()`**
  Returns general information about the grammar.

  **Arguments:**

  - None.

  **Returns:**

  - The output from the grammar.
    **Type:** `String`

- **`getLexRulesFromFile()`**
  Returns all lexical rules that are known to the grammar. This method calls the method `getPredefined()` with the argument `lexrules`.

  **Arguments:**

  - None.

  **Returns:**

  - The output from the grammar.
    **Type:** `String`.

- **`getLexemesFromFile()`**
  Returns all lexemes that are known to the grammar. This method calls the method `getPredefined()` with the argument `lilex`.

  **Arguments:**

  – None.

  **Returns:**

  – The output from the grammar.
    **Type:** `String`

- **`getFormsFromFile()`**
  Returns all words known to the grammar. This method calls the method `getPredefined()` with the argument `lifor`.

  **Arguments:**

  – None.

  **Returns:**

  – The output from the grammar.
    **Type:** `String`

- **`getPsRulesFromFile()`**
  Returns all rules that are known to the grammar. This method calls the method `getPredefined()` with the argument `rules`.

  **Arguments:**

  – None.

  **Returns:**

  – The output from the grammar.
    **Type:** `String`

- **`main()`**
  The main method of the class. Performs tests on the methods in the class. Note that this method is not supposed to be used for anything else than a test method!

  **Arguments:**

  – An array of arguments to the program
    **Type:** `Array of String`

  **Returns:**

  – Nothing.
    **Type:** `void`

### B.13 Class: PrologException

This class extends the class *Exception* and is used as a general exception for the application instead of the plethora of exceptions that *Jasper* uses.
**Methods:**

- **`PrologException()`**

### B.14 Class: Detail

This is a window used for showing the user details about things in the grammar.
**Methods:**

- **`Detail()`**
  The constructor for the class, creates the GUI.
- **`main()`**
  A main method, supposed to be used for testing purposes.

**Inner classes:**

- **`buttonListener`**
  Manages all actions generated by the button in the GUI.
- **`windowCloser`**
  Currently not used.