Reports from
Uppsala University
Department of Linguistics

RUUL
nr 36

Two Reports on Computational Syntax and Semantics

Mats Dahllöf

UPPSALA
UNIVERSITET

Institutionen för lingvistik

Januari 2003

Reports from
Uppsala University
Department of Linguistics

RUUL
nr 36

Two Reports on Computational Syntax and Semantics

Prolog-Embedding Typed Feature Structure Grammar (PETFSG-II) and Grammar Tool

An Implementation of Token Dependency Semantics for a Fragment of English

Mats Dahllöf

Uppsala universitet
Institutionen för lingvistik
Box 527
SE-751 20 UPPSALA, Sweden

Januari 2003
ISBN: 91-973737-2-9
ISSN: 0280-1337

RUUL (Reports from Uppsala University, Department of Linguistics), nr 36, Januari 2003.

WWW: `http://www.ling.uu.se/ruul/`.

Series editor: Åke Viberg.

Published by:
Uppsala universitet,
Institutionen för lingvistik,
Box 527,
SE-751 20 UPPSALA,
Sweden.

Typeset in LaTeX by the author.

ISBN: 91-973737-2-9 (this volume).
ISSN: 0280-1337 (*RUUL* series).

Printed by Universitetstryckeriet, Uppsala.

Report 1:

Prolog-Embedding Typed Feature Structure Grammar (PETFSG-II.2) and Grammar Tool

Report 2:

An Implementation of Token Dependency Semantics for a Fragment of English **101**

Report 1

Prolog-Embedding Typed Feature Structure Grammar (PETFSG-II.2) and Grammar Tool

Preface

A related (but in many respects very different) system by the same name was described in *RUUL* (Reports from Uppsala University Dept of Linguistics) **34**, 3–37, 1999. The grammar formalism and the system interface have been completely reworked here. Intermediary versions have been made available on the WWW.

URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/` is the site for a demo, updates on this system, source files, and related material.

I want to thank Roussanka Loukanova, who has used this grammar tool in her teaching in the department's course on computational grammar. Her comments have prompted me to improve several features of the system and its documentation. I am also grateful to Bengt Dahlqvist, Leif-Jöran Olsson, Per Starbäck, Anna Sågvall Hein, and Åke Viberg for relevant support.

Uppsala, December 2002

Mats Dahllöf (`mats.dahllof@ling.uu.se`)

1 Introduction

The present document describes a unification-based grammar formalism and a grammar tool for it. The formalism will be called 'Prolog-Embedding Typed Feature Structure Grammar II' (PETFSG). This is version II.2 of the package. The grammar tool is used over the WWW or locally in a Prolog interpreter (or in compiled form). Its main active component is a chart parser. The grammar tool allows Prolog procedures, so-called *postparse procedures*, to process the output from the parser before it is being displayed.

The PETFSG grammar tool is intended to be of value for both research and teaching purposes. PETFSG makes it easy to integrate Prolog components into the grammar. The grammar tool is furnished with an HTML-based user-friendly interface. The system provides facilities for displaying various components of the grammar and for tracing the chart-based application of the grammar in parsing. It also gives informative reports on errors encountered when the grammar is compiled.

The syntax of the PETFSG formalism is based on Prolog term syntax. Typed feature structures (FSs) are the basic means for encoding linguistic information. These allow embedding of ordinary (untyped) Prolog terms. Prolog terms may consequently be used whenever convenient, for instance, to make semantic representations graphically more compact, and in cases where the inventory of values is large or hard to predict (e.g. in semantics). Typing may at the same time be used to impose restrictions wherever useful. The PETFSG formalism also allows Prolog calls to be made from inside the grammar. In this way, Prolog computation can be used to take care of phenomena which can not be treated by means of FS unification. This makes the

PETFSG formalism powerful, flexible, simple, and easy to use. The PETFSG system runs under SICSTUS or SWI Prolog.

The PETFSG formalism and system have been designed with Head-Driven Phrase Structure Grammar (HPSG, Pollard & Sag 1994, Sag & Wasow 1999) in mind. The PETFSG system thus belongs to the same family of grammar tools as those associated with PATR-II (Shieber 1986), Attribute-Logic Engine (ALE, Carpenter & Penn 1997) and the Linguistic Knowledge Building (LKB) system (Copestake 2002).

The PETFSG system provides its own kind of user interface. The PETFSG formalism is based on Prolog and Prolog terms and this makes it easy to combine the grammar tool with various Prolog-based procedures. The type systems of PETFSG are more restricted than some of those used in the HPSG literature.

1.1 Basic Functions of a PETFSG Application

Before turning to more technical matters, a brief description of the facilities provided by the PETFSG grammar tool may be useful. Further details are given in Chapter 4.

[PS rules]**:** This link is used for listing the phrase structure rules.

[Forms]**:** This link is used for listing the surface word forms. Each word form is associated with a link that will display all FSs associated with the form in question.

[Lexemes]**:** This link works like the previous one, but for the lexemes.

[Lex. rules]**:** This link is used for listing the lexical rules.

[Types]**:** This link is used for displaying the type hierarchy.

[NFSs] (Named Feature Structures): This link is used for listing the FS macros that have been introduced in the grammar.

Parsing: A parsing form is used to parse strings according to the current grammar.

Task analysis: A task analysis form is used to request an analysis of what happens at a certain point in the (chart-based) parsing process.

Test suites: Test suites with a sequence of grammar tool requests may be defined. A link for each suite appears on the grammar tool pages and will cause the suite to be executed. This is useful for systematic testing of the grammar.

1.2 Components of a PETFSG Application

The PETFSG tool is primarily intended to be used over the WWW with a web browser as interface to the system. Grammar development is however a text-based process. A compiler is used to set up the resources needed for the web application. The linguistic and computational components behind a PETFSG application are given by the PETFSG Prolog code, by the application-specific PETFSG grammar (one or several files), and possibly by some further Prolog code, which provides definitions of the Prolog procedures required by the grammar. The grammar files also contain some further data concerning the application.

Local use of the PETFSG tool, not involving a web server, is also possible, see Section 5.6.

The source files behind a PETFSG application include the main PETFSG source code files (Prolog), the user's grammar files and additional Prolog files, and a cascading style sheet file. (Also see Appendix F.)

A PETFSG application is based on a number of resources. (See Chapter 5 for details on how to create these.)

- A simple CGI *script* is needed for the web interface. It can be of two kinds. It is either unique for a certain PETFSG applica-

tion or general for several PETFSG applications on the server end. The former kind of script can be automatically generated when a grammar is compiled.

- A Prolog *saved state* hosts the main PETFSG program components, the compiled grammar(s), and further Prolog definitions.
- Furthermore, a *cascading style sheet* should be used.

A PETFSG application requires SICSTUS or SWI Prolog and a web server on the server end, and a web browser on the client end. It seems that Netscape 6, Internet Explorer, and Mozilla are the best web browsers to use. FSs are represented as tables, often with many levels of embedding, and Netscape 4.XX (and lower) is unable to handle this. (Netscape 4.XX will only work properly provided that the FSs exhibited are shallow enough.)

1.3 User Requests

User requests transmitted over the WWW are handled by a CGI script, which calls the Prolog system into which the application saved state file thereby is loaded. As mentioned above, there are two possible ways of doing this: The identity of the application saved state file may be hard-coded into the (automatically generated) CGI script file. We can also use a CGI script which is general for several applications (saved state files). This method is described in Section 5.7. The general presentation will presuppose that the former kind of CGI interface is used.

The content of a user request is given by a number of key values. An overview of these is given in Appendix A.

The PETFSG system produces HTML output. The CGI script and Prolog processes terminate when a user request has been fulfilled.

2 PETFSG Grammars

A PETFSG grammar exists in the form of a sequence of Prolog terms, each delimited by a full stop. (For details on Prolog syntax, the reader is referred to the relevant Prolog documentation.) These Prolog terms will be referred to as *grammar items*. A grammar is placed in a text file or in a sequence of text files.

Several grammars may be handled within a single PETFSG application. An overview of how to do this is given in Section 2.15.

The grammar file (or sequence of files) is compiled into an internal form by the PETFSG system. During the compilation the internal coherence of the grammar is checked, and error messages are issued whenever there is something wrong with type or feature declarations and whenever grammar rules and lexical entries fail to agree with them. Contradictions within grammar items are also reported. Faulty items in the grammar are otherwise ignored by the system (i.e. not stored in the internal database).

A PETFSG application uses two lexical databases, one for lexemes and one for surface word forms. The surface word forms are those that occur as constituents of actual linguistic expressions. The lexemes are entities to which the lexical rules can be applied, thereby producing new lexemes or surface word forms. The lexical rules are also allowed to operate on surface word forms. The distinction between the two kinds of lexical entity is defined by the *leaf symbol definition* (see below).

A PETFSG grammar contains grammar items as follows: (The various kinds of item are discussed in more detail below.)

- Precisely one *type and feature declaration*.
- At most one *initial symbol definition*.
- At most one *leaf symbol definition*.
- Any number of *named* FS *(*NFS*) definitions*.
- Any number of *phrase structure rules*.
- Any number of *lexical item entries* (each of which is added to the lexeme or surface form lexicon according to the leaf symbol definition).
- Any number of *lexical rules*.
- Any number of *regular definitions of morphological functions*.
- Any number of *suppletive forms* (exceptions to the the regular definitions of morphological functions).

Apart from grammar items, grammar files will also contain the following kinds of item:

- There will be precisely one *application identifier clause* for each PETFSG application.
- Current grammar declarations may appear anywhere.
- A number of *interface information terms* may also occur.

2.1 The Application Identifier Clause

An *application identifier clause* is of the form `grammar(`*Name*`,`*Dir*`)` or `application(`*Name*`,`*Grammars*`,`*Dir*`).` `grammar(`*Name*`,`*Dir*`)` declares that *Name* is the name of the application. The atom[1] *Dir* is

[1]The term *atom* is used here in the sense of textual constant, as in the documentations for SICSTUS and SWI Prolog.

the directory in which the application saved state is to be stored. This is an example:

```
grammar('sampleII.2','/home/staff/matsd/petfsg/').
```

The name of the application saved state file will be derived from *Name* (by the extension of `.sav` or `.swi`, depending on whether SICSTUS or SWI Prolog is used). In this case the saved state file will reside as `/home/staff/matsd/petfsg/sampleII.2.sav` or `/home/staff/matsd/petfsg/sampleII.2.swi`.

A term of the form `application(`*Name*`,`*Grammars*`,`*Dir*`)` is used as application identifier clause when the application is to contain more than one grammar. Here, *Grammars* is the list of grammar identifiers. This is an example:

```
application('sampleII.2',
            [eng,swe],
            '/home/staff/matsd/petfsg/').
```

Here, `eng` and `swe` are introduced as grammar identifiers. The first one, in this case `eng`, is the default grammar. When a `grammar/2` application identifier clause is used, the application will host only one grammar.

An error message will appear if the compiler encounters grammar items at a position where the current grammar has not been defined.

The application identifier will appear on the output pages. So will the identifier for the current grammar in case there are several grammars in the application.

2.2 Current Grammar Declarations

A term of the form `current_grammar(`*Grammar*`)` declares *Grammar* to be the current grammar when subsequent grammar items are compiled. Its scope extends to the next such term, or for the rest

of the compilation process. The default grammar will be the current grammar until this state is changed by means of a current grammar declaration clause.

2.3 Declaring the Types and Features

A term of the form `declaration` *TH* `where` *FL* provides a declaration of the type hierarchy and the features associated with the types as described here.

A PETFSG type system is a simple type hierarchy without cross-classification (i.e. a tree structure). Every type is thus the immediate subtype of at most one type. The type system is based on a set of named most general types. Features may be associated with any type of the hierarchy.

The type system is defined by a list of terms, which are of the form *Type* `subsumes` *Subtypes*. Here, *Type* is an atom which names a type and *Subtypes* is a list of `subsumes/2`-terms. Such a term defines the type *Type*. The terms in *Subtypes* define the set of immediate subtypes to *Type*. The `subsumes`-terms of *Subtypes* are interpreted in the same way. The most specific types, i.e. those without subtypes, are consequently defined by terms of the form *Type* `subsumes []`. This notation allows us to represent the entire type hierarchy as a list of `subsumes`-terms.

Features are associated with (user-defined) types by means of a list of feature declarations, each of which is of the form *Type* `features` *Features*, where *Type* is a type and *Features* is a list of feature-type pairs. A feature-type pair is of the form *Feature*:*Type*, where *Feature* is a Prolog atom naming a feature and *Type* is the type of the values of this feature. Every feature name is in this way associated with precisely two types. First, there is the type that carries the feature, in the sense that the feature is appropriate for this type and all its subtypes. Secondly, there is the type of the value of the feature.

If there is no feature declaration for a type, *Type*, no feature is as-

sociated with it. A term of the form *Type* `features []` would make this explicit.

In the grammar item `declaration` *TH* `where` *FL*, *TH* is the type hierarchy (a list of `subsumes`-terms, as described above) and *FL* is the list of feature declarations (a list of `features`-terms, as described above). There is precisely one such `declaration` term in a PETFSG grammar, and it can only be preceded by the application identifier clause.

A very simple sample grammar (to a large extent due to Sag & Wasow 1999 and listed in Appendix C) will be used for illustration throughout. This is its type hierarchy declaration (cf. Sag & Wasow 1999: 386). (Readability is improved by a suitable indentation.)

```
declaration
    [ss_struc subsumes [
       phrase subsumes [],
       lex_item subsumes [
         word subsumes [],
         lxm subsumes [
           const_lxm subsumes [],
           infl_lxm subsumes []]]],
     agr_cat subsumes [
       '3sing' subsumes [],
       non_3sing subsumes []],
     gram_cat subsumes [],
     pos subsumes [
       verb subsumes [],
       adv subsumes [],
       adj subsumes [],
       comp subsumes [],
       conj subsumes [],
       prep subsumes [],
       nom subsumes [
```

```
          noun subsumes [],
          det subsumes []]],
      sem_struc subsumes [],
      form_type subsumes [
        fin subsumes [
          pres subsumes [],
          pret subsumes []],
        inf subsumes [
          infin subsumes []]],
      mode_type subsumes [
        none subsumes [],
        prop subsumes [],
        ques subsumes [],
        dir subsumes [],
        ref subsumes []],
      boolean subsumes [
        '+' subsumes [],
        '-' subsumes []]]
where
   [ss_struc features
      [syn:gram_cat,
       sem:sem_struc],
    gram_cat features
      [head:pos,
       comps:list,
       spr:list,
       gap:list],
    pos features
      [form:form_type,
       pred:boolean],
    nom features
      [agr:agr_cat],
    det features
```

```
    [count:boolean],
  sem_struc features
    [mode:mode_type,
     index:prolog_term,
     restr:prolog_term]].
```

In addition to the user-defined types provided by the `declaration` term, PETFSG recognizes `list` and `prolog_term` as two types. The data structures of the grammar will, as previously mentioned, be called *feature structures* (FSs). They are consequently of three kinds: *attribute-value* FS*s*, which are values of the user-defined types, *lists*, and *Prolog terms*.

2.4 Representing Types

The PETFSG formalism uses Prolog atoms to represent PETFSG types as follows.

- Types from the user-defined type hierarchy are represented by their names.
- The Prolog term type is represented by the name `prolog_term`.
- The name of the type of lists is `list`.

2.5 Specifying FSs: Value Terms

The expressions that are used to characterize PETFSG FSs will be called PETFSG *value terms*. The following list defines the syntax of PETFSG value terms. Their semantics will be treated below. A PETFSG value term is one of the following:

- a PETFSG *variable*, which is any Prolog atom beginning with an `x`, e.g. `xIndex`, or

- a *type name*, i.e. any Prolog atom representing a type, as described in Section 2.4, or
- a *:-expression*, which is a Prolog term of the form *Feat*:*Val*, where *Feat* is a feature name (i.e. a Prolog atom) and *Val* is a PETFSG value term, or
- a `<>`-*term*, which is a Prolog term of the form `<>`*X*, where *X* is any Prolog term, or
- a $\{\ldots\}$-*term*, which is a Prolog term of the form $\{T_1,\ldots,T_n\}$ ($\{\}$ being a special case), where T_1, ..., T_n are PETFSG value terms, or
- a `^`-*term*, which is a Prolog term of the form *H* `^` *T*, where *H* and *T* are PETFSG value terms (*T* should stand for a PETFSG list), or
- an `nfs`-*term*, which is a Prolog term of the form `nfs` *X*, where *X* is a Prolog atom, or
- a `[...]`-*term*, which is a Prolog term of the form $[T_1,\ldots,T_n]$, where T_1, ..., T_n are PETFSG value terms.

These are a few examples of PETFSG value terms:

- ```
  phrase
  ```
- ```
  restr: <>[named(I,'Kim')]
  ```
- ```
  [word,
   syn:[head:verb,
        spr:{[nfs np,
              sem:[index: <>I]]},
        comps:{[nfs np,
                sem:[index: <>J]]}],
   sem:[mode:prop,
  ```
  ```

```
            index: <> S,
            restr: <>[love(S,I,J)]]]
```

- ```
  [syn:[head:[agr:xAgr],
        spr:{[syn:[head:[agr:xAgr]]]}]]
  ```

I will not provide a formal semantics for PETFSG value terms, but the principles below should make their use clear. A PETFSG value term is always understood as standing for an instance of the most general FS compatible with the explicit requirements of the expression. The various kinds of PETFSG value term are understood as follows:

- A PETFSG variable stands for any PETFSG FS. Co-occurrence of variables indicates sharing of substructure, as usual. The scope of a PETFSG variable extends over one grammar item.

- A type name stands for an FS which is of the type in question.

- A :-expression, *Feat*:*Val*, represents an FS (compatible with the current declarations) in which the feature *Feat* carries the value represented by *Val*. For instance, `agr:3sing` characterizes an FS in which the `agr` feature carries the value `3sing`. (It would be of type `nom` given the declarations of the sample grammar.)

- A <>-term, <>*X*, is treated like an ordinary Prolog term (with respect to unification). The scope of Prolog variables embedded in such terms extends over one grammar item.

- A $\{\ldots\}$-*term*, $\{T_1,\ldots,T_n\}$, represents a PETFSG list whose items are the FSs represented by the PETFSG value terms $T_1, \ldots, T_n$, arranged in this order. (A special case is that the PETFSG value term $\{\}$ stands for an instance of the empty PETFSG list.)

- A ^-*term*, *H* ^ *T*, stands for a PETFSG list having *H* as its head and *T* as its tail. *T* is required to represent a PETFSG list.
  ```

- An `nfs`-*term*, `nfs` *X*, is equivalent to the PETFSG value term that has been given the name *X* (in an `is_short_for` item). An error is reported if no NFS by the name of *X* has been defined. (Two different instances of `nfs` *X* will represent two different, but equivalent, FSs.) See below, Section 2.8.

- A `[...]`-*term*, $[T_1,\ldots,T_n]$, represents the (most general) FS subsumed by each of the FSs represented by the PETFSG value terms $T_1,\ldots,T_n$. (This allows us to write complex PETFSG value terms which are similar to the bracketed matrices that are commonly used in presentations of feature structure grammars.)

2.6 Initial Symbol Definition

A grammar item of the form `initial_symbol` *ValExpr*, where *ValExpr* is a PETFSG value term, defines the initial symbol to be the FS which is the value of *ValExpr*. Only instances of this FS are considered to be possible descriptions of syntactically isolated linguistic expressions, i.e. counted as parses by the parser. If an initial symbol definition is lacking, then any FS is counted as a parse by the parser. The absence of an initial symbol definition is consequently equivalent to the presence of this one: `initial_symbol []`.

The initial symbol definition from the sample grammar:

```
initial_symbol
   [syn:[head:[form:fin],
         spr:{},
         comps:{}]].
```

2.7 Leaf Symbol Definition

A Prolog term of the form `leaf_symbol` *ValExpr*, where *ValExpr* is a PETFSG value term, defines which lexical items are taken to be possible descriptions of surface word forms. As mentioned above, other

lexical items serve as lexemes from which other lexemes and surface word forms are derived by means of lexical rules. If a leaf symbol definition is lacking, then any FS is counted as a possible surface word form. The absence of a leaf symbol definition is thus equivalent to the presence of this one: `leaf_symbol []`.

The sample grammar contains this leaf symbol definition:

```
leaf_symbol word.
```

2.8 Named FS (NFS) Definitions (Macros)

A specification of an FS may be named and thereafter reused. This is useful in cases where the same complex structure appears several times in the grammar. A structure is given a name by means of a grammar item of the form *Name* `is_short_for` *ValExpr*. *ValExpr* is a PETFSG value term.

This is an example (from the sample grammar):

```
saturated_noun is_short_for
  [syn:[head:noun,
        spr:{},
        comps:{}]].
```

Given this definition, an instance of `nfs saturated_noun` is equivalent to the PETFSG value term occurring as the second operand to `is_short_for`.

2.9 Phrase Structure Rules

The PETFSG phrase structure rules allow Prolog calls to occur in the right-hand sequence. The left-hand item of a rule is a PETFSG value term (corresponding to the mother node). The right-hand term is a list, each of whose members is of one of the following two kinds:

- A PETFSG value term matches a single ordinary syntactic daughter (the standard kind of right-hand item in grammar formalisms, in other words).

- A term of the form `prolog [`P`, `A_1`, ..., `A_n`]`, where P is any Prolog term and $A_1, \ldots, A_n$ is a sequence of PETFSG value terms (typically PETFSG variables), causes the parser to make the Prolog call `call_prolog([`P`, `V_1`, ..., `V_n`])`, where V_1 is the FS value associated with the PETFSG value term A_1, and so on, as internally represented by the Prolog system. (See Section 3.7 on how to deal with these values.) In this way, the predicate `call_prolog/1` provides the interface between the phrase structure rules and general Prolog computation. Every solution to the `prolog` call (or sequence of such calls) will be considered by the parser, i.e. result in a separate edge of the chart. (It should consequently have a finite number of solutions.) The intention is that P should identify a procedure, whose arguments are given by the $A_1, \ldots, A_n$ values. Examples are found in the quoted rules below.

A rule is of the form *Name* `rule` *LH* `===>` *RH*. *LH* is the left-hand PETFSG value term. *RH* is the right-hand list of daughter PETFSG value terms and `prolog` calls. *Name* is an atom naming the rule.

Returning to our sample grammar, we find the following two rules:

```
head_complement rule

   [phrase,
    syn:[head:xHead,
         comps:xComps,
         spr:xSpr],
    sem:[mode:xMode,
         index:xIndex,
         restr:xRM]]
                    ===>
```

```
   [[syn:[head:xHead,        % head
          comps:xComp^xComps,
          spr:xSpr],
     sem:[mode:xMode,
          index:xIndex,
          restr:xR1]],

    [xComp,                  % complement
     sem:[restr:xR2]],

    prolog [compute_semantics,xR1,xR2,xRM]].

specifier_head rule

   [phrase,
    syn:[head:xHead,
         comps:{},
         spr:{}],
    sem:[mode:xMode,
         index:xIndex,
         restr:xRM]]
                    ===>
   [[xSpr,                   % specifier
     sem:[restr:xR1]],

    [syn:[head:xHead,        % head
          comps:{},
          spr:{xSpr}],
     sem:[mode:xMode,
          index:xIndex,
          restr:xR2]],

    prolog [compute_semantics,xR1,xR2,xRM]].
```

The rule `head_complement` combines a head with the first complement on its COMPS list. (One complement is taken care of by each application of the rule.) The rule `specifier_head` assembles a specifier and a head. In both rules, when the two constituents have been found by the parser, Prolog is called by the query `call_prolog([compute_semantics,` V_1`,` V_2`,` V_3`])`, where V_1, V_2, and V_3 are the FSs associated with the PETFSG variables `xR1`, `xR2`, and `xRM` (in that order). (Section 3.7 describes how feature values are accessed.) The `call_prolog/1` procedure for the sample grammar is defined by the following clause:

```
call_prolog([compute_semantics,
             Semantics1,
             Semantics2,
             Semantics3]):-
         compose_semantics(Semantics1,
                           Semantics2,
                           Semantics3).
```

Here, the Prolog procedure `compose_semantics/3` is simply called with the three given arguments.

2.10 Lexical Entries

A lexical entry is of the form *Form* `>>>` *ValExpr*. *Form* is the term representing the textual form of the word. The system assumes that textual forms are given as Prolog atoms. *ValExpr* is the PETFSG value term defining the FS associated with the entry. If the FS is compatible with the leaf symbol definition, it is stored in the surface form lexicon; if it is not, it is stored in the lexeme lexicon.

The following lexical entries (from the sample grammar) provide some illustration:

```
a >>>
```

```
  [word,
   syn:[head:[det,
              agr:'3sing',
              count:'+'],
        spr:{},
        comps:{}],
   sem:[mode:none,
        index: <> '-',
        restr: <>[]]].

'Kim' >>>
  [const_lxm,
   nfs saturated_noun,
   syn:[head:[agr:'3sing']],
   sem:[mode:ref,
        index: <> I,
        restr: <>[named(I,'Kim')]]].
```

2.11 Lexical Rules

New lexical entries may be derived by means of HPSG-style lexical rules. A PETFSG lexical rule involves two FS specifications (in the form of PETFSG value terms) a morphological function name (an atom, see Section 2.12), and possibly also a Prolog call. A lexical rule is thus of the form `lexrule` *Name* `morph` *Morph* `input` *In* `output` *Out* [`prolog` *Call*], where *Morph* is a morphological function and *In* and *Out* are PETFSG value terms representing an input and an output structure. The term *Call* in the optional `prolog` *Call* part, is a Prolog call, made in the same way as those in the phrase structure rules. Whenever the *In* value unifies with the FS description on a lexical entry, the rule 'generates' a new entry from the resulting *Out* value of the lexical rule. Structure sharing between the two feature structures of a lexical rule indicates how the information from the input entry is

rearranged into a description of the new entry. Note that there is no default passing of information.

When a lexical rule with a `prolog` *Call* part is applied, the call *Call* is made and every solution results in a new lexical entry. [2]

The textual form of a new entry produced by the application of a lexical rule is given by the morphological function, *Morph*, which is applied to the textual form of the input entry to produce the textual form of the output item. (Morphological functions are further described in Section 2.12.)

A lexical rule may, for instance, look like this one in the sample grammar:

```
lexrule verb_pres_3sing
  morph  v_third_singular_infl
  input  [infl_lxm,
          syn:[xSyn,
          head:verb],
          sem:xSem]
  output [word,
          syn:[xSyn,
          head:[form:pres],
          spr:{syn:[head:[agr:'3sing']]}],
          sem:xSem].
```

This rule derives a singular agreement present tense surface word form from a verb lexeme.

The application of the lexical rules is triggered when the grammar files have been compiled by means of a `compile_grammar/1` command, as described in Section 5.1.

[2] `lexrule` *Name* `morph` *Morph* `input` *In* `output` *Out* is thus synonymous with `lexrule` *Name* `morph` *Morph* `input` *In* `output` *Out* `prolog true`.

2.12 Morphological Functions

The PETFSG view of morphology is fairly simplistic. It supports an HPSG-style concept of *morphological function* (Sag & Wasow 1999). A morphological function is defined by associating it with an *atom reshaping function* (defined in Prolog) for the regular (non-suppletive) cases and a number of statements concerning suppletive forms. Suppletive form statements take precedence over a regular case statement.

A morphological function is associated with an atom reshaping function by means of a term of the form `infl_reg(`*Morph*`,`*AR*`)`. Here, *Morph* is the morphological function and *AR* the atom reshaping function. For instance:

```
infl_reg(sing_infl,id).
infl_reg(plur_infl,affx(s)).
infl_reg(v_third_singular_infl,affx(s)).
```

The last of these clauses define the morphological function involved in the lexical rule cited in Section 2.11.

An atom reshaping function, identified by an arbitrary Prolog term, is defined *directly in Prolog*, by means of the predicate `atom_reshaping/3`, relating an atom reshaping function identifier, an input textual form, and an output textual form, as described in Section 3.2.

2.13 Suppletive Forms

Statements concerning suppletive forms are of the form `suppletive_form(`*Form1*`,`*Morph*`,`*Form2*`)`. Such a clause states that the morphological function called *Morph* gives the suppletive form *Form2* as output when applied to the textual form *Form1*. This is an example:

```
suppletive_form(child,plur_infl,children).
```

Now, a suppletive form statement takes precedence in the sense that `infl_reg` clauses are applied only when there is no `suppletive_form` clause which is applicable.

2.14 Interface Information Terms

The interface information terms provide pieces of information which concern technical aspects of a PETFSG application rather than its linguistic substance. It is however convenient to store them in the PETFSG grammar files, as this interface information concerns the location of the application resources, the treatment of the grammar output, and its visual appearance. (There is no checking of the correctness of the interface information terms of the kind performed on the proper grammar components.)

Cascading style sheets: A term of the form `css_url(`*URL*`)`, *URL* being an atom, defines the URL for a cascading style sheet (CSS) for the output pages. This is not necessary, but the use of a CSS may improve the graphical appearance of the output pages.

FS **colours**: A term of the form `type_color(`*Type*`,`*Colour*`)` defines the output colour associated with the type *Type* (which should appear in the type hierarchy). *Colour* is a standard HTML colour specification, i.e. a hexadecimal number prefixed by a hash mark, e.g. `'#0000FF'`, in the form of a Prolog atom. A default colour is used if there is no explicit information about the colour of a type. A `type_color(`*Type*`,`*Colour*`)` declaration does not apply to the subtypes of *Type*. Possible example: `type_color(infl_lxm,'#FA8072')`. The use of `type_color` declarations is optional.

Postparse procedures: As mentioned above, the grammar tool allows Prolog procedures, so-called *postparse procedures*, to modify the output from the parser. These procedures are identified by means of Prolog atoms. A term of the form `postparse_procedures(`*List*`)` declares the set of atoms that are names of available postparse proce-

dures, these atoms being listed in *List*. The first item will be the default choice. Possible example: `postparse_procedures([sem_only])`.[3] See Section 3.4.

Test suites: A term of the form `suite(`*Name*`,`*List*`)` defines a test suite associated with the application. The atom *Name* is used to identify the suite and *List* is a list of test suite commands. A test suite command is a list of key-value pairs of the form *Key*=*Value*. These correspond to the key-value pairs with which a PETFSG application CGI script is called (see Appendix A and the figures). However, values of the `str` key are given as ordinary Prolog lists. This is an example:

```
suite(s1,[[act=types],
          [act=word, str='walks'],
          [act=parse, str=['Kim', walks],
           a1=nomod, a2=no],
          [act=parse, str=['Kim', loves, a, dog],
           a1=sem_only, a2=no],
          [act=parse, str=[a, dog, walks],
           a1=nomod, a2=yes]]).
```

This suite, identified by `s1`, displays the type hierarchy and the *walks* lexical entries, and performs parsing with three different settings of the key values. So, for instance, `[act=parse, str=['Kim', loves, a, dog], a1=nomod, a2=no]` is a parsing request, where `['Kim', loves, a, dog]` is the string to be parsed, `nomod` is the postparse procedure, and `no` chart overview is to be printed. (It is equivalent to the request of Figure 4.7.)

Start page section: It is also possible to specify a section of HTML matter that will appear on the application start page. It can be used to provide information about the application, e.g. links to relevant documentation. This text is supplied in a term of the form `start_page_section(`*T*`)`, where *T* is a list of Prolog atoms. These

[3]The predicate `postparse_procedure/3`, which forms the interface to the general Prolog system, must be defined accordingly. See Section 3.4.

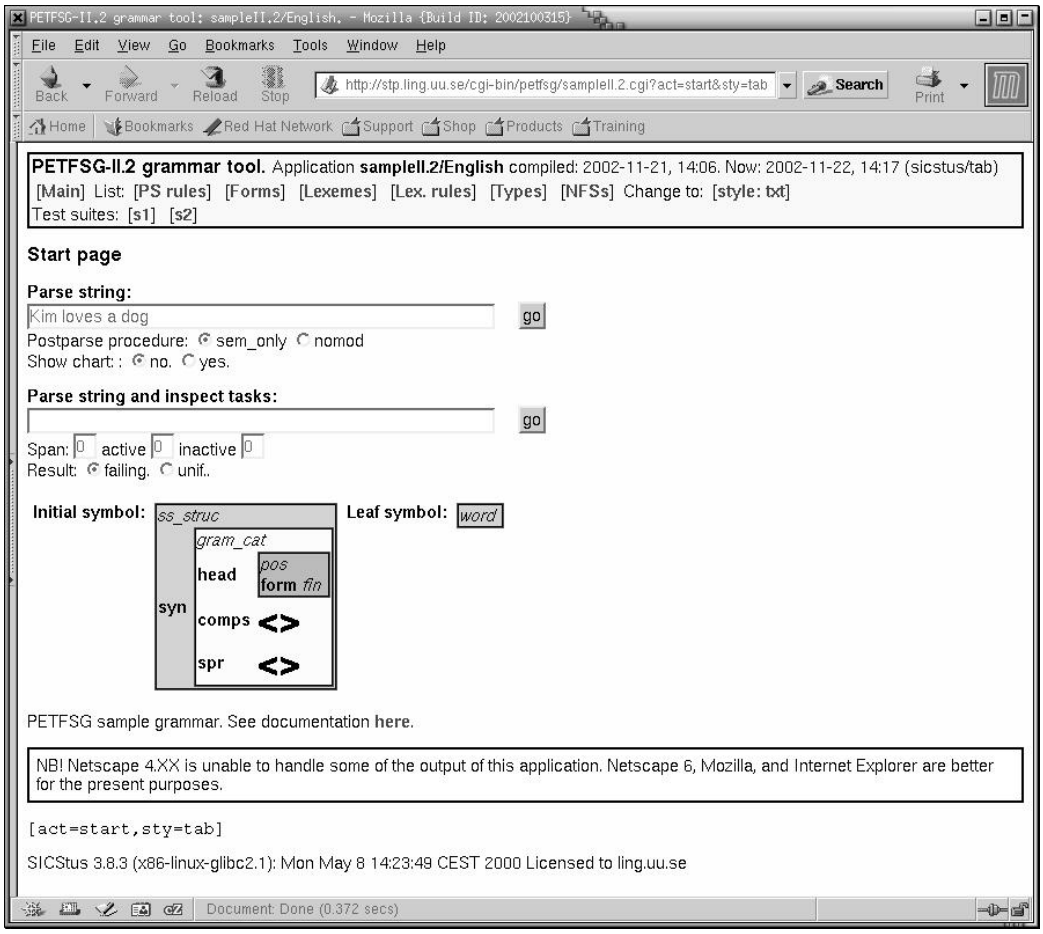


Figure 2.1: The start page. The request-specific key-value string is: `act=start`.

will be printed, one after the other, in the context of an HTML paragraph. This example produces an information paragraph as shown in Figure 2.1.

```
start_page_section([
   '<p>PETFSG sample grammar. See ',
   'documentation <a href="http://',
   'stp.ling.uu.se/~matsd/petfsg/II.2/">',
   'here</a>.</p>']).
```

Only one `start_page_section(`*T*`)` term can be used.

User-defined grammar tool acts: Additional values for the request key `act` can be defined by means of the Prolog procedure `call_user_petfsg_act/2` (see Section 3.6) The atoms which form possible `act` values to be treated in this way are defined by a term of the form `user_act_values(`*Atoms*`)`, where *Atoms* is a list of those atoms.

2.15 Management of Multiple Grammars

The PETFSG grammar tool treats grammars as modules according to a simple scheme. At each point in a grammar file being compiled, a certain grammar is the current grammar. The compiler counts encountered grammar items as belonging to the current grammar. The management of multiple grammars may be summarized as follows:

- The set of grammars are defined by an application identifier clause of the form `application(`*Name*`,`*Grammars*`,`*Dir*`)`, where *Grammars* is the list of grammar identifiers.
- The *first* item in the *Grammars* list above is the *default* grammar. It is the current grammar unless another one is explicitly selected.
- A current grammar declaration of the form `current_grammar(`*Grammar*`)`, where *Grammar* is a grammar identifier, makes *Grammar* the current grammar.

In the user requests, the grammar is selected by means of the `gram` key, see Section 4.7.

3 Interfaces to Prolog Procedures

A PETFSG application is allowed to make use of general Prolog procedures in a few contexts. The Prolog file associated with the sample grammar is listed in Appendix D. It provides examples of the mechanisms described in this section. These Prolog procedures, which belong to a certain application, but are accessed from within the PETFSG system, are placed in a Prolog module by the name of `this_application`. It is declared in this way (the public predicates being described below):

```
:- module(this_application,
          [call_prolog/1,
           atom_reshaping/3,
           label_token/3,
           postparse_procedure/3,
           dsp_prolog_term/4,
           call_user_petfsg_act/2]).
```

The sample grammar source files are organized in such a way that these Prolog procedures are loaded into the module `this_application`, from the file `this_grammar('application_procedures.pl')`, as required in the file `library('this_application.pl')`.

3.1 Prolog Calls from Rules

Prolog calls from the phrase structure rules are made by means of the predicate `call_prolog/1`, as described in Section 2.9.

3.2 Atom Reshaping Functions

As mentioned in the section on morphological functions, 2.12, an atom reshaping function is identified by an arbitrary Prolog term. The function is then defined directly in Prolog, by means of the predicate `atom_reshaping/3`. It relates an atom reshaping function identifier, an input textual form, and an output textual form. It should be called with the first two arguments bound, and give one solution with the third argument bound to an atom.

For instance, the following Prolog clauses define the morphology needed in Section 2.12: `id` is identity, and `affx(`*A*`)` adds affix *A*, atoms[1] being used to represent words:

```
atom_reshaping(id,Atom,Atom).

atom_reshaping(affx(Affix),Atom_in,Atom_out):-
    !,
    name(Atom_in,Atom_in_name),
    name(Affix,Affix_name),
    append(Atom_in_name,
           Affix_name,
           Atom_out_name),
    name(Atom_out,Atom_out_name).
```

3.3 Token Labeling

The predicate `label_token/3` is called just before a lexical inactive edge is stored in the chart. Its arguments are the FS, the number giving the position of the word in the input string (an integer), and its textual form (an atom). This allows us to associate a unique label with each input word token. If this feature is not used, `label_token/3` has to be

[1]The predicate `name/2` is the relation between a Prolog atom and the list of characters defining its textual form.

defined by `label_token(_,_,_)`, i.e. as always succeeding without effects. (This is the case in the sample grammar.)

3.4 Postparse Procedures

The predicate `postparse_procedure/3` defines Prolog procedures which are applied to the output FSs after the parsing process. The first argument is the name (probably a Prolog atom) identifying a procedure. The second argument corresponds to the input FS and the third argument to the list (set) of FSs that will be associated with the input FS by the procedure. A `nomod` procedure is, so to speak, available by default. It corresponds to the definition: `postparse_procedure(nomod,FS,[FS])`. The `nomod` procedure consequently does not perform any computation at all. User-defined postparse procedures should be declared by `postparse_procedure` terms, as described in Section 2.14.

This is an example of a `postparse_procedure/3` clause:

```
postparse_procedure(sem_only,ALL,[SEL]):-
        !,
        path_value(ALL,sem,SEM),
        path_value(SEL,sem,SEM).
```

The `sem_only` procedure gives us, for each analysis, an FS for which only the `sem` value is defined and this value is identical to that of the original FS.

3.5 Prolog Term Printing

A special predicate, `dsp_prolog_term/4`, is responsible for displaying Prolog terms embedded within PETFSG objects. This feature is documented in Section 4.4.

3.6 Application-Specific Grammar Tool Acts

The procedure `call_user_petfsg_act/2` is used to define application-specific acts for a PETFSG application. Its first argument is an `act` key value atom, which must be among those defined by the application's `user_act_values/2` term (see Section 2.14). The second argument is the list of argument key-value pairs of the form used in test suite definitions (see Section 2.14), e.g. `[act=new_parse, str=['Kim', loves, a, dog], a1=sem_only, a2=no]`.

3.7 Accessing Feature Values in FSs

Prolog calls of the kind described in Sections 2.9 and 3.1 necessitate a device for accessing feature values in FSs, as internally represented. The predicate `path_value/3` (belonging to the module `fs_handling`) is useful for this purpose. `path_value(`*FS*`,`*Path*`,`*Val*`)` holds when *FS* is a given internal-form FS value, *Path* is a sequence of feature atoms joined by the infix colon functor, and *Val* is the value embedded within *FS* which the path *Path* identifies. For instance, `path_value(`*FS*`,f:g:h,`*Val*`)` holds if *Val* is the value of the feature `h` in the value of the feature `g` in the value of the feature `f` in the FS *FS*, both *FS* and *Val* being FSs as internally represented by the PETFSG system. (The `path_value/3` device contributes to hiding the internal representation.) Examples are found in Section 3.4 and in Appendix D.

4 Using a PETFSG Application

This section describes in some detail how a PETFSG application is used. (Chapter 5 describes how to create an application.) The application start page appears when the CGI script for the application is called from the browser with the key-value pair `act=start`. Its appearance is shown in Figure 2.1.

Each output page has the same top 'panel' carrying information and links of general relevance. The application identifier (here, `sampleII.2`) is printed along with information about when the grammar was compiled and when the page was created. When the application hosts more than one grammar, the identifier for the current grammar is printed together with the application identifier. The panel also carries links which list various kinds of grammar item, links for test suites, and forms for parsing and task analysis. The link [Main] produces the start page. [Main/txt] or [Main/tab] also produces the start page, but changes the FS display style (see Section 4.4). There are also links for the non-current languages in case the application is multilingual.

4.1 Listing Grammar Items

Various items of the grammar can be listed. The links which are used for such requests are found in the top panel.

- The phrase structure rules are listed by means of the [PS rules] link. Each rule is printed vertically, as in Figure 4.1: First we find the left-hand item, then the items of the right-hand list. Prolog

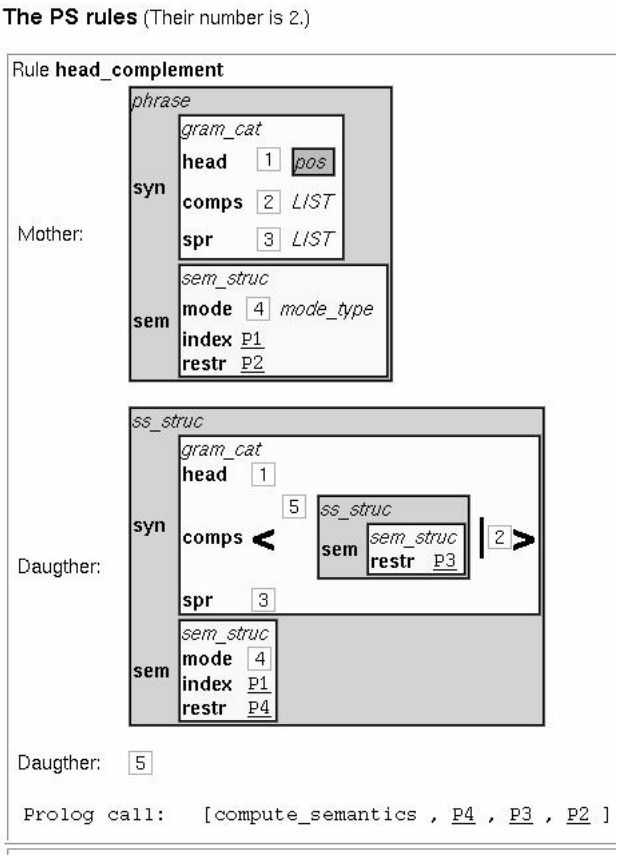


Figure 4.1: Phrase structure rule listing (cut). The request-specific key-value string is: `act=rules`.

call items are identified by the phrase `Prolog call` followed by the Prolog call, with the feature structures passed as arguments displayed.

- The [Forms] link produces a listing of all surface word forms present in the current application, as in Figure 4.2. A number is also associated with each textual form. It gives the number of surface form entries for the textual form in question. Each listed item is associated with a link that will display all lexical entries (both surface forms and lexemes) associated with the textual form. (The [Lexemes] link produces a list of all lexemes in

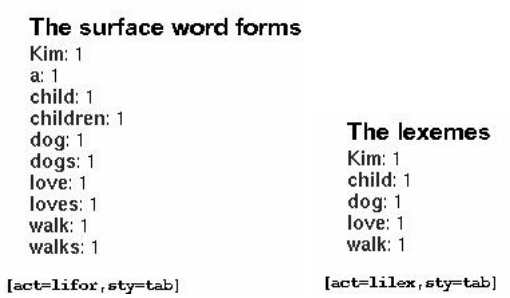


Figure 4.2: Listings of surface word forms and of lexemes. The request-specific key-value string is: `act=lifor` and `act=lilex`, respectively.

a corresponding way.) A word in the lexicon is displayed as in Figure 4.3.

- The [Lexemes] link produces a listing of all lexemes present in the current application, as in Figure 4.2. A number is also associated with each one, giving the number of lexeme entries for the textual form in question. Each listed item is associated with a link that will display all lexical entries (both lexemes and forms) associated with the textual form, as in the [Forms] case.

- The [Lex. rules] link produces a listing of the lexical rules. The morphological function, the input pattern, and the output pattern are displayed for each rule, as in Figure 4.4. If there is a Prolog call in the rule, it is displayed in the style of the phrase structure rules. The passing of information from the input FS to the output FS, i.e. the way in which they share substructures, is shown by means of a sequence of equations.

- The [Types] link produces an overview of the current type hierarchy and, to the right, the features associated with the various types. The tree structure is shown by means of indentation, as in Figure 4.5.

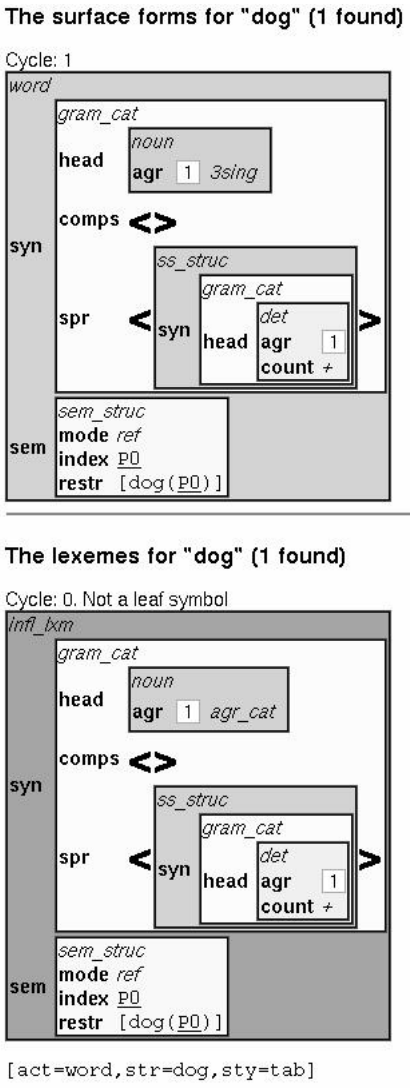


Figure 4.3: A lexical entry. The request-specific key-value string is: `act=word&str=dog`.

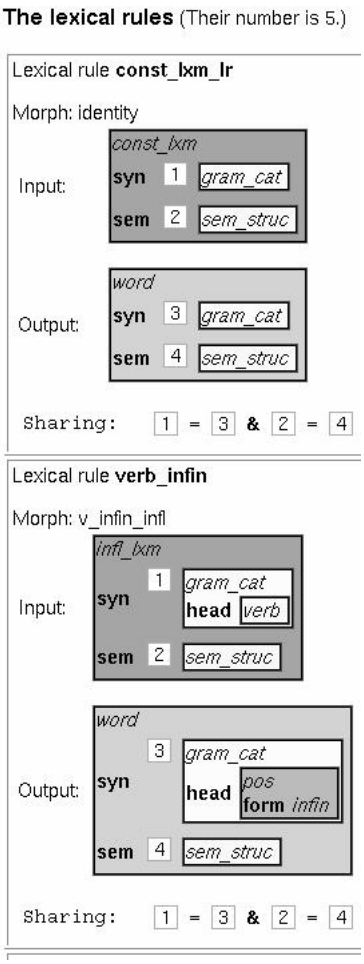


Figure 4.4: Lexical rule listing (cut). The request-specific key-value string is: `act=lexrules`.

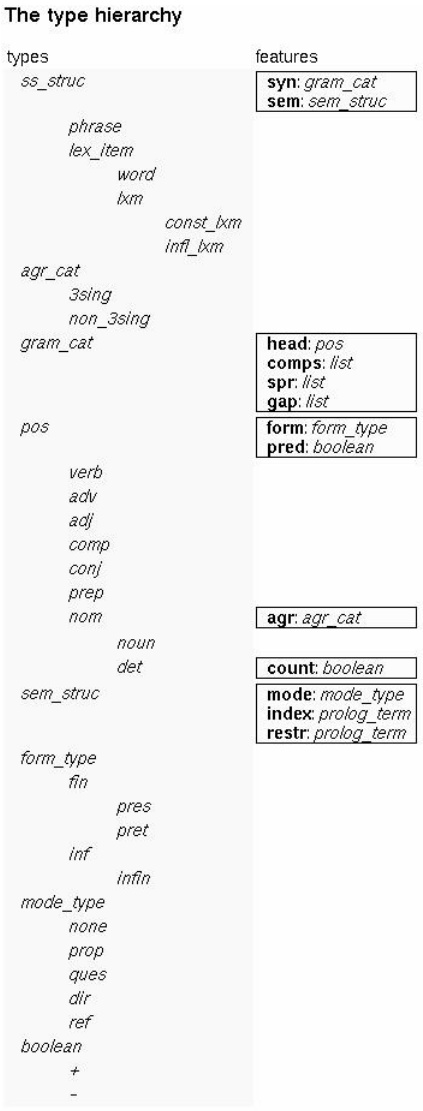


Figure 4.5: Type hierarchy (the sample grammar). The request-specific key-value string is: `act=types`.

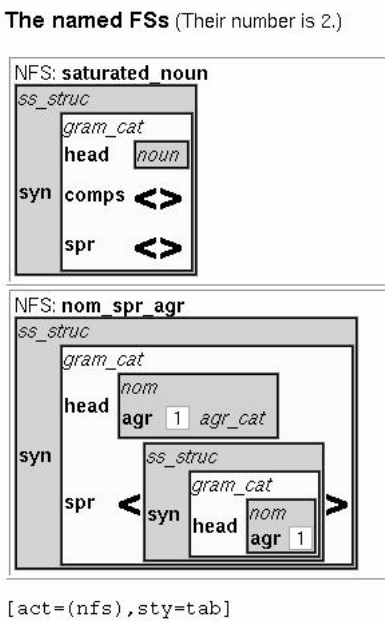


Figure 4.6: Listing of named feature structures. The request-specific key-value string is: `act=nfs`.

- The named feature structures are listed by means of the [NFS] link. An example is found in Figure 4.6.

The left hand column also contains test suite links (as specified by the `suite` interface information terms).

4.2 The PETFSG Tool Parser

The PETFSG parser allows us to inspect how the grammar works or fails to work, by applying it to strings. The parser is a simple predictive chart parser. The input string is given to the parser as a sequence of Prolog atoms. When the parsing form on the PETFSG start page (Figure 2.1) is used, input is entered as running text and is tokenized before being sent to the parser. (More about this in Section 4.3.)

As usual, complete and incomplete analyses are stored in *edges*, and the locations of words and substrings are defined by pairs of nodes.

A node is an integer: 0 is the beginning of the input string. 1 separates the first word from the second, etc.

Inactive edges represent completely analyzed substrings. An inactive edge is defined by a start node, an end node, and an FS (linguistic description).

Active edges represent partially recognized expressions. An active edge is defined by a start node, an end node, an FS (linguistic description), and an action list. The action list defines what is required to complete the assemblage of an expression. Actions correspond to the right-hand side items in grammar rules. They are consequently of two kinds: those corresponding to not-yet-found daughters and `prolog` calls.

An important operation of the parser is *rule invocation*: A phrase structure rule defined by a grammar term of the form *Name* `rule` *LH* `===>` *RH* is said to be *invoked* at a node, *n*, whenever an active edge with *n* as start and end node, a description corresponding to *LH*, and an action list corresponding to *RH* is added to the chart. This means that the parser initiates the assemblage of a constituent according to the rule in question, starting at the node *n*.

Now, the parsing process proceeds as follows, given an input string in the form of a list of Prolog atoms of length L.

- The first step of the parser is to perform lexical analysis. For each n, such that $1 \leq n < L$, it adds one inactive edge, with $n-1$ as start node and n as end node, for each description of the n'th atom in the input list found in the grammar's surface word form lexicon. If the first word has an uppercase initial letter, the corresponding lowercase atom is also matched against the lexicon. For instance, an initial token of *Three* will match a *three* in the lexicon.

- The second step is to invoke *every* grammar rule at the first node (0) of the chart.

- The parsing process in a chart parser consists in the evaluation of tasks. A task is a pair of an active and inactive edge that meet, in the sense that that the end node of the active edge is identical to the start node of the inactive edge. The tasks that are detected when new edges are added to the chart are put in the *agenda*, where they wait for evaluation. The agenda is treated as a queue in the PETFSG parser.

  A task is evaluated in the traditional fashion (i.e. in accordance with the 'fundamental rule' of chart parsing): If the first FS, D_A, on the action list (of the active edge) is incompatible with the FS of the inactive edge, D_I, the task is evaluated without effects. In the opposite case, i.e. if the two FSs, D_A and D_I, are compatible, they are unified. If there is one or several Prolog calls following the FS D_A on the action list, these are evaluated, possibly resulting in several solutions. Each one of these produces a new edge which is stored in the chart. The action list on these contains those items which remain after the D_A item and the Prolog calls have been removed from the action list.

  When a new active edge is added, the parser invokes, at the end node of the active edge, each PS rule whose right-hand FS is compatible with the first FS on the action list on the active edge, i.e. each rule which can possibly generate a constituent of the expected kind starting at the end node of the new active edge. (This represents a top-down prediction strategy.)

- The parsing process terminates when all tasks have been evaluated.

A grammar may cause the parser to loop. This happens when the grammar assigns an infinite number of analyses to some substring of the input string. Needless to say, this situation should never be produced by a 'useful' grammar.

4.3 The Parsing Form

A parsing form is found on the PETFSG grammar tool start page. See Figure 2.1. A text input field is used for the string to be parsed. Radio buttons are used for the selection of a postparse procedure (one of the user-defined ones or nomod). If the yes/no radio button is pressed yes an overview of the chart is exhibited. The form is called by means of a go submit button. The input text is then converted into a string of atoms. Each substring of characters without blanks counts as one token, the blanks being interpreted as separators. Lower- and uppercase is preserved. As mentioned above, if the first word has an uppercase initial letter, the corresponding lowercase atom is also matched against the lexicon.

The result of a request from the parsing form is displayed on a separate page. Each FS which is associated with an inactive edge spanning the whole input string and which is an instance of the initial symbol is considered to represent a parse. Each parse FS is modified according to the choice of postparse procedure (unless the choice is nomod). An overview of the chart is also given, if this choice was selected in the parsing form. The parse request in Figure 2.1 gives us the output in Figure 4.7, given the sample grammar.

When the yes alternative of the Show chart radio buttons is selected (`a2=yes` in the request-specific key-value string), the chart is exhibited as a table with one row for each span, i.e. pair of nodes, defining a segment of the input string. The corresponding substring is displayed along with the number of inactive and active edges having the span in question. Each number is associated with a link that produces a listing of the edges concerned. See Figure 4.9 for an example.

4.4 How the FSs are Displayed

There are two styles available for the display of FSs. One is based on indentation and the use of a fixed width font. The other and more

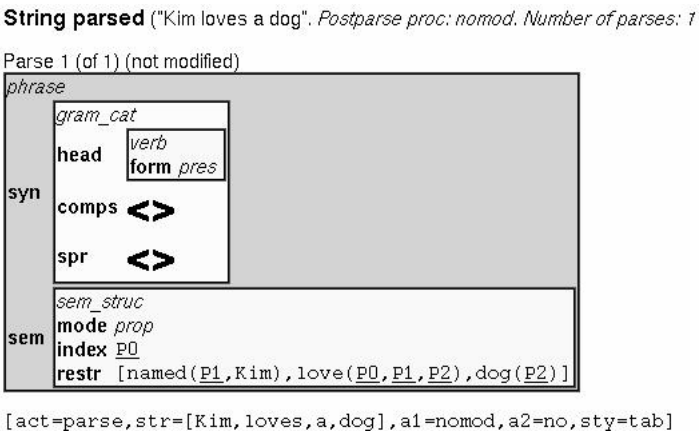
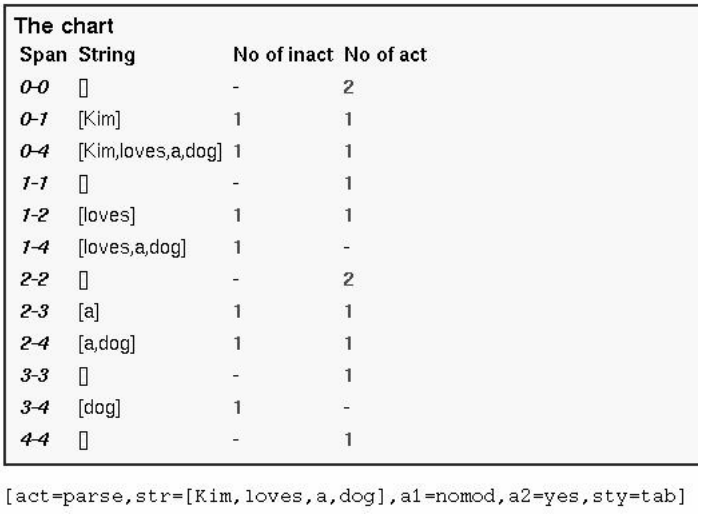


Figure 4.7: A parse. The request-specific key-value string is: `act=parse&str=Kim+loves+a+dog&a1=nomod&a2=no`.



The chart

Span	String	No of inact	No of act
0-0	[]	-	2
0-1	[Kim]	1	1
0-4	[Kim,loves,a,dog]	1	1
1-1	[]	-	1
1-2	[loves]	1	1
1-4	[loves,a,dog]	1	-
2-2	[]	-	2
2-3	[a]	1	1
2-4	[a,dog]	1	1
3-3	[]	-	1
3-4	[dog]	1	-
4-4	[]	-	1

Figure 4.8: A chart overview. The request-specific key-value string is: `act=parse&str=Kim+loves+a+dog&a1=nomod&a2=yes`.

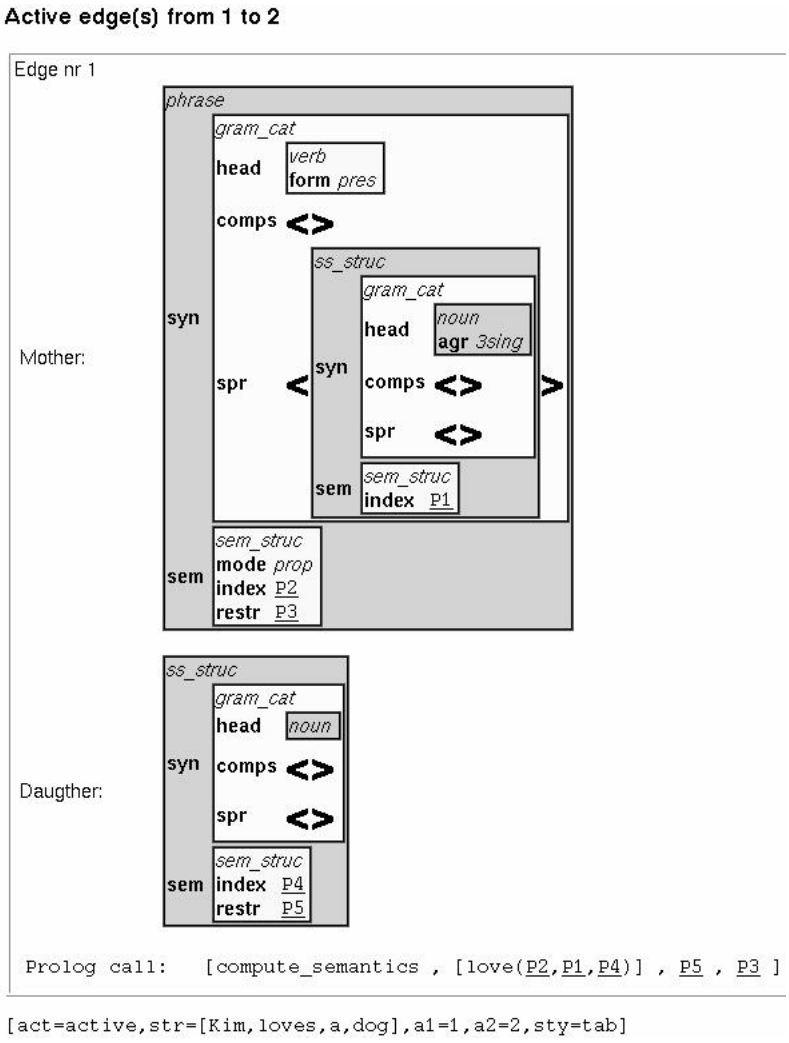


Figure 4.9: An edge in the chart. In this context, Daughter means a constituent which is required to be found by the edge. The request-specific key-value string is: `act=active&str=Kim+loves+a+dog&a1=0&a2=1`.

elegant style uses framed tables. The two styles are associated with the identifiers `txt` (text) and `tab` (tables). The display style is specified by means of the request key value `sty` (see below).

The table display style (`sty=tab`) is as follows: Sharing of substructure (re-entrancy) is shown by means of boxed numbers, as is the common practice. Re-entrancy is not shown in case of empty PETFSG lists. In Prolog terms (see below), re-entrancy is shown by means of Prolog variables.

A numeral enclosed within a rectangle (and highlighted by means of colour) is used to refer to a substructure the first time it occurs, in which case it is displayed. This numeral comes directly to the left of the FS. Subsequent instances of the same boxed numeral indicate where the same value re-enters the FS.

The FSs of (user-defined) types which allow features are displayed as framed tables, i.e. as boxes. (Square brackets are not used.) Otherwise, the FS typography is similar to the standard way of displaying FSs. The type appears in the upper left corner. Below, the features, to the left, are coupled with their values, to the right. When a value is completely underspecified the feature-value-pair is not printed, unless it is required to indicate re-entrancy. Completely underspecified FSs are in this case displayed as *ANY*. It is thus possible that there are no feature-value-pairs shown in a printed FS.

The `type_color` terms of the grammar determine the colours associated with the different (user-defined) types. A default colour is used if there is no `type_color` term for the type involved.

A completely underspecified PETFSG list appears as *LIST*. An empty PETFSG list is displayed as <>. Otherwise, PETFSG lists appear with their elements arranged horizontally, '<' marking the beginning and '>' the end of the list. Commas separate the items. An underspecified tail is preceded by a vertical bar ('|'), as in the Prolog notation for lists.

The text display style (`sty=txt`) uses indentation to exhibit the levels of embedding within the FSs. Numerals enclosed within bars

(‘|’) serve as boxed numerals. Feature names appear capitalized.

As mentioned briefly in Section 3.5, Prolog values appear as printed by the predicate `dsp_prolog_term/4`. It is responsible for the displaying of Prolog terms embedded within PETFSG objects in both `txt` and `tab` style. This procedure allows us to define application-specific printing conventions, for instance, to improve readability. The first argument is the Prolog term to be displayed. The second one is the current indentation as a natural number giving the distance in character positions from the left. (This is needed in `sty=txt` display of FSs.) The third and fourth arguments, also natural numbers, are used to provide numbered and easy to read symbols for the Prolog variables. The third argument is the input counter value and the fourth one is the output counter value, which will be the input counter value to the subsequent `dsp_prolog_term/4` call for the same grammar item. (The input counter argument is 0 the first time the predicate is called in the context of a grammar item.)

This definition gives us plain `write/1`:ing of Prolog terms with easy-to-read variables.

```
dsp_prolog_term(Term,
                _,
                Variable_number_in,
                Variable_number_out):-
    instantiate_variables(Term,
                          Variable_number_in,
                          Variable_number_out),
    write(Term).
```

The Prolog terms are simply `print/1`-ed in HTML ’<tt>’ style. The `instantiate_variables/3` call gives us a new numbering of the variables (see the file `application_procedures.pl` in Appendix D). (The indentation argument of `dsp_prolog_term/4` is not used here.)

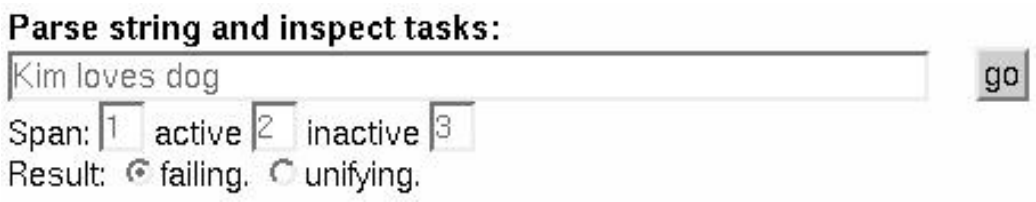


Figure 4.10: The task analysis form.

4.5 Task Analysis

The parser stores the chart parsing tasks that have been evaluated. This makes it possible to trace what has happened during the parsing process. The PETFSG grammar tool thus provides a task analysis facility, which can be used for grammar analysis and debugging purposes. The task analysis form carries four text input fields, one for the string to be parsed, and three for node numbers. The first of these is the start node of the active edge; the second one is the end node of the active edge, which is also the start node of the inactive edge; and the third node number is for the end node of the inactive edge. Radio buttons selecting either unif. or fail determine whether successful or failing tasks are to be listed. The form is called by means of a go submit button. Figure 4.10 shows an example.

The active edge in a task is associated with an action list, whose head is the FS which corresponds to the constituent that is to be found. The task succeeds if this FS can be unified (i.e. is compatible) with the description on the inactive edge. A task which is successful in the present sense may however fail to produce new edges. This happens if a subsequent Prolog call on the action list fails (i.e. produces no solutions). A task fails (in the present sense) if and only if the head FS on the action list is incompatible with the description on the inactive edge. In the case of failing tasks, the task analysis facility gives us an explanation of each unification failure. The failure explanation tells us where the contradiction causing the unification failure is located. This

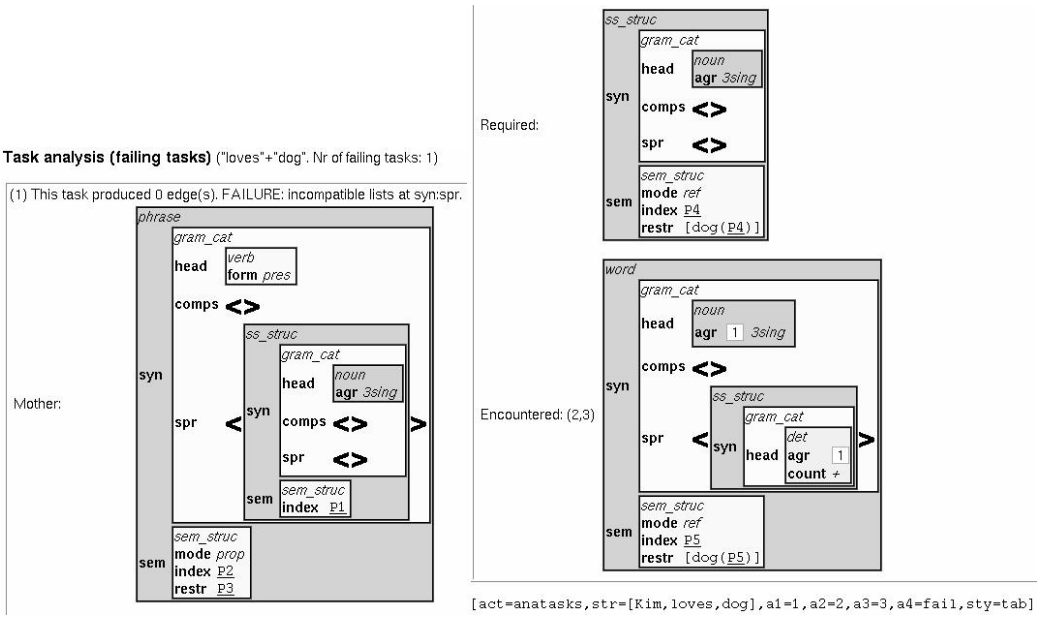


Figure 4.11: A task analysis report (one column layout on screen). The request-specific key-value string is: `act=anatasks&str=Kim+loves+dog&a1=1&a2=2&a3=3&a4=fail`.

location is described by means of a path expression. Terms of the form `LI(`*n*`)` are used as feature names to refer to the *n*th item of a list. For instance, `spr:LI(1):head:number` means the `head:number` value of the first item of the `spr` list. See Figure 4.11.

4.6 Running a Test Suite

A link for each test suite is found on the top panel, as in Figure 2.1. A suite is executed when the corresponding link is selected. The suite output page gives us the output for each item in the suite, arranged vertically and framed in boxes. The CPU time is also measured. It is reported at the end of the output page.

4.7 Selecting Grammar

In case the application contains several grammars, the grammar is selected by means of the `gram` key. The phrase structure rule listing in Figure 4.1 could also have been requested by means of the request-specific key-value string `act=rules&gram=English`, even if the `gram` key is redundant for the monolingual sample grammar.

5 Creating an Application

A PETFSG WWW application is based on two files residing on the server end, the CGI script and the saved state file. The use of a CSS file is also recommended. File names for the program state and for the CSS file are given by the `grammar/2` or `application/3` and `css_url/1` terms found in the grammar files. The permissions on these files must be set according to their use here. This is done automatically for the CGI script and saved state file.

The files for the PETFSG grammar tool are collected under URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/pgt/`, and those for the sample grammar under URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/sample/`.
The PETFSG grammar tool program is divided into a number of modules and corresponding files. The module names are listed and the module contents briefly described in Appendix F. The files `sicstus.pl` and `swi.pl` contain procedures whose definitions depend on which variety of Prolog is used. (One of them should be copied onto the file `this_prolog.pl`.) The system has been tested under SICSTUS Prolog version 3.8.3 and SWI Prolog version 5.0.2. The web application is based on a Shell CGI script and has been tested under an Apache (version 1.3.23) web server.[1]

[1]The CGI key-value pairs are passed through the environment variable `QUERY_STRING` from the web server to the Prolog process.

5.1 Creating the Saved State (SICSTUS)

This section describes how the application-specific saved state is created in SICSTUS Prolog. Section 5.3 describes how the same thing is done when SWI Prolog is used.

The PETFSG grammar files must be converted into the internal form used by the grammar tool by means of the predicate `compile_grammar/1` (belonging to the module `compiler`). In the process, the grammar is also checked for consistency. An overview of compilation error messages is given in Appendix B. The argument to `compile_grammar/1` is a list of atoms giving the names of the files in which the grammar resides. The grammar may be dispersed over any number of files. A `compile_grammar/1` call removes the previous grammar(s), if an application is already loaded into the Prolog/PETFSG system.

The grammar files are compiled in the order in which they appear in the list argument to `compile_grammar/1`. The type and feature `declaration` must be compiled before the other grammar items. A named FSs must be defined before it is referred to. Error messages and warnings are issued when contradictions and errors occur in grammar items (cf. Appendix B).

The lexical rules are used to expand the set of available lexical entries. They are not applied during the parsing process. This lexical inflation process is triggered by the `compile_grammar/1` command (after the grammar files have been compiled, needless to say). As a first step, each lexical rule is matched against every explicitly given lexical entry. Then, there is a second cycle of application, in which the lexical rules are applied to the new entries produced in the first cycle. Each time a lexical rule matches a lexical entry a new lexical entry is added to the lexicon, or one for each Prolog call solution, if there is such a part in the rule. New lexical entries are not compared to the already present ones before being added. The lexical rules are then repeatedly matched against the items produced in the previous cycle.

This iteration terminates when no new items are produced in a cycle of application. The grammar writer should make sure that the lexical rules do not produce infinite loops.

The procedure `save_grammar_state/0` is used to save the PETFSG system and the compiled grammar data base as a Prolog saved state file.

A convenient way of creating the saved state from a grammar is to consult a file defining the necessary steps. For the sample grammar application, as used under SICSTUS Prolog, one with the following content has been used. (The file is called `make_sample.pl`):

```
:- assertz(user:library_directory(
           '$HOME/lib_petfsg/II.2/pgt')).
:- assertz(user:file_search_path(
           this_grammar,
           '$HOME/lib_petfsg/II.2/sample')).
:- use_module(library(compiler)).
:- compile_grammar(
          [this_grammar('gram_sample.pl')]).
:- save_grammar_state.
:- make_cgi_script(
          '/local/etc/httpd/cgi-bin/petfsg/').
```

The two first items define the locations for the PETFSG grammar tool program files, and the grammar files, respectively. The file `gram_sample.pl` is the sample grammar used for illustration here. The call `:-save_grammar_state` saves the present Prolog state as a saved state file with the name specified by the grammars's `grammar/2` term. The `make_cgi_script` call creates a CGI script file, and is further discussed below.

The module `this_prolog` must contain a few procedures whose definitions are different in SICSTUS and SWI Prolog. The Prolog procedures that belong to the present application are loaded into the module `this_application`, from the file

`this_grammar('application_procedures.pl')`, as required in the file `library('this_application.pl')`. (These Prolog procedures are described in Chapter 3.)

The PETFSG system reports on the compilation in the following fashion. (Lines have been broken manually here.)

```
--- PREPARING COMPILATION ---.

--- COMPILING: this_grammar(gram_sample.pl) --- .
Compiled declaration.
initial_symbol/leaf_symbol/head_complement/
specifier_head/saturated_noun/a/Kim/const_lxm_lr/
verb_infin/verb_pres_3sing/walk/love/nom_spr_agr/
3sing_noun/non_3sing_noun/agent_noun/dog/child/
Expanding English lexicon from level 0.
const_lxm_lr:1/verb_infin:2/
verb_pres_3sing:2/3sing_noun:2/
non_3sing_noun:2/

Expanding English lexicon from level 1.

{/home/staff/matsd/petfsg/sampleII.2.sav
 created in 15 msec}

--- SAVED GRAMMAR STATE IN sampleII.2.sav ---.
Created the CGI script
/local/etc/httpd/cgi-bin/petfsg/sampleII.2.cgi
{consulted
/home/staff/matsd/lib_petfsg/II.2/make/
  make_sample.pl
in module user, 570 msec 370448 bytes}
```

The names of the compiled items are reported. The numbers of successful applications of the lexical rules are also reported. The `chmod`

operations set the appropriate permissions on the created files.

This process presupposes that the module `this_prolog` gives us a few SICSTUS-specific procedures. They are defined in the file `library('sicstus.pl')`. When SICSTUS Prolog is used, the file `library('sicstus.pl')` is simply copied onto `library('this_prolog.pl')`.

If the file `this_grammar(application_procedures.pl')` contains definitions of the `this_application` procedures used by the sample grammar, these must be put in this module by means of the file `library('this_application.pl')`. It has the following content, which can remain unchanged over several applications.[2]

```
:- module(this_application,
          [call_prolog/1,
           atom_reshaping/3,
           label_token/3,
           postparse_procedure/3,
           dsp_prolog_term/4,
           call_user_petfsg_act/2]).

:- compile(this_grammar(
           'application_procedures.pl')).
```

5.2 The Application CGI Script (SICSTUS)

As mentioned, this call is responsible for the creation of an appropriate CGI script file.

```
:- make_cgi_script(
        '/local/etc/httpd/cgi-bin/petfsg/').
```

[2]The file name expansions, as specified by the dynamic predicate `file_search_path/2`, may be changed (see the file `make_sample.pl`, above).

The CGI script resides in the server end file system at a suitable location (according to the requirements of the web server). The argument to `make_cgi_script/1` defines the directory in which the file is placed.

The following (bash shell) CGI script (the file `sampleII.2.cgi`) will be produced for the sample grammar application.

```
#!/bin/sh
echo -e "Content-Type: text/html\n\n"
/local/bin/sicstus -r /home/staff/matsd/petfsg/sampleII.2.sav
exit
```

Here, the shell is defined on the first line. The content type is declared. Then SICSTUS Prolog is started from the application saved state. In a new environment, the content of the CGI script has to be modified according to the local situation.[3]

The name of the saved state file and the directory in which it is stored are derived from the grammar's `grammar/2` or `application/3` term, which in this case may look as follows:

```
grammar('sampleII.2','/home/staff/matsd/petfsg/').
```

As mentioned above, the saved state file name is created by adding the extension of `.sav` to the application name, if SICSTUS Prolog is used.

5.3 Using SWI Prolog

A grammar-specific saved state is created in SWI Prolog in much the same way as in SICSTUS Prolog, but a few differences should be noted: As can be expected, the file `library('swi.pl')` is used instead of `library('sicstus.pl')`

[3] The procedures `make_cgi_script/1` in `compiler.pl` and `make_prolog_call/1` in `sicstus.pl` may have to be modified.

under SWI Prolog. So, `library('swi.pl')` is copied onto `library('this_prolog.pl')`.

The main steps for creating the application are also somewhat different. This proposal seems to work, but I cannot say that it is the best solution to the problem it solves. In case of the sample grammar application they are the following. (This is the SWI Prolog counterpart, `make_sample_swi.pl`, to the file `make_sample.pl`.)

```
:- assertz(user:library_directory(
    '/home/staff/matsd/lib_petfsg/II.2/pgt/')).
:- assertz(user:file_search_path(
    this_grammar,
    '/home/staff/matsd/lib_petfsg/II.2/sample/')).
:- consult(library('operators.pl')).
:- use_module(library(compiler)).
:- use_module(library(main_petfsg),
              [start_call/0]).
:- compile_grammar([
'/home/staff/matsd/lib_petfsg/II.2/sample/gram_sample.pl']).
:- save_grammar_state.
:- make_cgi_script(
    '/local/etc/httpd/cgi-bin/petfsg/').
```

The call `:- consult('operators.pl')` makes all operator declarations available in the `user` module. The file `operators.pl` is simply the collection of all operator declarations also found in the various PETFSG module files. Furthermore, `:- use_module(main_petfsg,[start_call/0])` makes the main `start_call/0` available in the `user` module. The other steps are as for the SICSTUS Prolog case. This way of making an application causes the SWI interpreter to issue error messages when it encounters `use_module` declarations inappropriate for SWI, but this procedure will work anyway.

The CGI script for the sample grammar application in SWI Prolog

looks like this:

```
#!/bin/sh
echo -e "Content-Type: text/html\n\n"
/usr/bin/pl -x /home/staff/matsd/petfsg/sampleII.2.swi -t start_call
exit
```

This script is only different from its SICSTUS counterpart in the main command line, where SWI Prolog is started up. The saved state file name is, as mentioned above, created by adding the extension of `.swi` to the application name, if SWI Prolog is used.

5.4 Step by Step Instruction

Let me give a step by step description of how to create a new PETFSG application, which includes a summary of previous points. (The single CGI script method is described in Section 5.7.) In order to build a new application from the sample grammar files and the `make_sample.pl` (for SICSTUS) file or `make_sample_swi.pl` file (for SWI), do the following:

- File names may be changed according to the user's preferences. (The 'old' file names are used here.)
- Decide whether to use SICSTUS or SWI Prolog. Copy `sicstus.pl` or `swi.pl` onto the file `this_prolog.pl` depending on this choice.
- Give the grammar application a name. Modify the `application/3` item of the grammar file `gram_sample.pl` accordingly.
- Decide in which directory to put the saved state file (and create the directory, if necessary). Modify the `grammar(`*Name*`,`*Dir*`)` item of the grammar `gr_sample.pl`, where *Dir* is this directory. Files in this directory must be readable by web server processes.

- Decide in which directory to put the CGI script (and create the directory, if necessary). Modify the `make_cgi_script(`*Dir*`)` term, where *Dir* is this directory, in the file `make_sample.pl`.
- Decide on a WWW location for the CSS file, and put it there. The `css_url` term in the grammar must define the URL of the CSS.
- Consult the file `make_sample.pl`/`make_sample_swi.pl` from the selected Prolog interpreter. This should create all the files necessary to run the application, i.e. the CGI script and the saved state file.

5.5 Running an Application

The start page is requested by means of an URL calling the CGI script with the key-value pairs `act=start`. The `act` key is used to define the main kind of request sent to the script. An application based on the sample grammar is thus called by means of the URL: `http://stp.ling.uu.se/cgi-bin/petfsg/sampleII.2.cgi?act=start`.

This request produces the start page from which the options described in Chapter 4 may be selected by means of links and forms. `http://stp.ling.uu.se/cgi-bin/petfsg/sampleII.2.cgi` is the application-related part, while `act=start` is the request-specific key-value list. Other `act` values necessitate further request-specific key values, e.g. as in Figures 4.3, 4.7, 4.8, 4.9, and 4.11. (Consult Appendix A for an overview of the various possible combinations.)

5.6 Local Use of the PETFSG Tool

The PETFSG grammar tool can also be used locally in a Prolog interpreter. The predicate `petfsg_to_file/2` handles a PETFSG request and stores the HTML result in a file. `petfsg_to_file/2` can be called

after the compilation of the grammar. Its first argument is the request-specific key values, in the form of a Prolog list in which each element is a pair of the form *Key*=*Value*. As in suite commands, `str` values are given as ordinary Prolog lists. The second argument to `petfsg_to_file/2` is the name of the output file. This file should then be viewed in a web browser. These are a few examples of the use of `petfsg_to_file/2`.

```
petfsg_to_file([act=lifor],
               'op.html').

petfsg_to_file([act=parse,
                str=['Kim',loves,a,dog],
                a1=nomod,a2=no],
               'op.html').

petfsg_to_file([act=active,
                str=['Kim',loves,a,dog],
                a1=0,a2=1],
               'op.html').

petfsg_to_file([act=anatasks,
                str=['Kim',loves,a,dog],
                a1=1,a2=1,a3=3,&a4=fail],
               'op.html').

petfsg_to_file([act=suite,a1=s1],
               'op.html').
```

The first four calls correspond to Figures 4.2, 4.7, 4.9, and 4.11, respectively.

5.7 Using a General CGI Script (SICSTUS)

Another method for executing user requests involves a general CGI script, which may be used with any number of different saved state files. The saved state file is then selected by a CGI key value, ‘`gf`’, for ‘grammar file’. This method is only available in the SICSTUS version of the system. It involves the use of a start-up saved state, which is responsible for loading the application saved state file, as requested by the `gf` key value.

The general PETFSG start-up SICSTUS *saved state* (which is necessary if a general CGI script is used) provides an interface between the general CGI script and the PETFSG application. The CGI script has to reside on the server end file system according to the requirements of the web server. The following (bash shell) CGI script (the file `suII.2.cgi`) can be used for the sample grammar application.

```
#!/bin/sh
echo -e "Content-Type: text/html\n\n"
/local/bin/sicstus -r /home/staff/matsd/petfsg/sampleII.2.sav
exit
```

Here, the shell is defined, at the first line. The content type is declared. Then SICSTUS Prolog is started from the start-up saved state. In a new environment, the content of the CGI script has to be modified according to the local situation.

A start-up SICSTUS saved state is created when the file `start_up_petfsg.pl` file is consulted or compiled. It is stored in the file referred to in the CGI script and identified by the `save_program` clause in `start_up_petfsg.pl`.

6 Concluding Remarks

The PETFSG grammar tool has not been evaluated in any systematic way, but a few comments on its performance can be made. It has been used in both research and teaching. It has continually been developed and bugs have been fixed. It is my experience that the system now is in a robust condition.

Among the attractive features of the PETFSG grammar formalism and tool are the following:

- The interface makes it easy to apply the grammar in parsing and to get an overview of what it contains.
- The Prolog-oriented nature of the grammar formalism and the way in which the grammar tool allows Prolog procedures to be integrated into grammar applications may be attractive to Prolog users.
- The task analysis mechanism is a powerful tool for exploring the way in which a grammar works, very useful for the grammar writer when the grammar fails to produce intended analyses. It is also valuable in teaching situations.
- The test suite facility allows the grammar writer to examine the consequences of grammar modifications in a systematic way.

Of course, the PETFSG system does not deliver everything that a user of constraint-based grammar tools might wish for. The following potential shortcomings may be mentioned:

- There is no support for multiple inheritance.

- There is no support for any notion of principle as general as that of e.g. Sag & Wasow (1999).
- There is no default passing of information in the lexical rules. This makes the grammar writer's life somewhat more complicated. It also makes the rules more difficult to read. However, the more explicit passing of information may be more perspicuous from a pedagogical point of view.
- An attractive interface for local use of the grammar tool is lacking. The present one, described in Section 5.6, represents a somewhat *ad hoc* solution.

The system seems to be reasonably time efficient. In general the PETFSG grammar tool takes less time producing its result than the web browser takes to exhibit it, given the grammars that have been developed. The PETFSG grammar has not been designed for large scale use. The parsing algorithm is of a non-sophisticated kind, as is its implementation in Prolog.

The PETFSG formalism provides a straightforward and flexible means for implementing HPSG style grammars. The system has proved useful for teaching purposes, as it gives the students a way of exploring the grammars and provides help in grasping what happens when a grammar is put to use.

The next part of the present volume gives an example of how the PETFSG system can be used in computational semantics. The system has proved especially useful in contexts where there is a need for combining parsing with external computation in an explorative fashion.

References

Carpenter, B. & G. Penn, 1997, *The Attribute Logic Engine: User's Guide*, Version 2.0.3, Carnegie Mellon University, Philosophy Dept.

Copestake, A., 2002, *Implementing Typed Feature Structure Grammars*, CSLI Publications, Stanford.

Sag, I.A. & T. Wasow, 1999, *Syntactic Theory: A Formal Introduction*, Stanford, CSLI Publications.

Pollard, C. & I. A. Sag, 1994, *Head-Driven Phrase Structure Grammar*, Chicago & London, The University of Chicago Press.

Shieber, S.M., 1986, *An Introduction to Unification-Based Approaches to Grammar*, Stanford, CSLI.

Appendix A: Calling a PETFSG Application

A PETFSG application is called with a number of CGI key values. The '`gf`' key value defines the application saved state file when a general CGI script is used (see Section 5.7). The '`sty`' key value specifies the style of FS display, '`tab`' and '`txt`' being the two possible values. '`tab`' is the default when an explicit '`sty`' value is lacking. The '`gram`' key value decides which grammar is used in case the application contains several grammars. When no '`gram`' value is given, the default grammar is used.

The other ones, in the table below, define the action requested from the PETFSG application.

The '`str`' key value defines the linguistic form or string which the request concerns.

act	str	a1	a2	a3	a4
start	-	-	-	-	-
rules	-	-	-	-	-
lifor	-	-	-	-	-
lilex	-	-	-	-	-
lexrules	-	-	-	-	-
types	-	-	-	-	-
nfs	-	-	-	-	-
suite	-	name of suite	-	-	-
word	atom	-	-	-	-
parse	list	name of postparse	yes/no (chart)	-	-
active	list	node number	node number	-	-
inactive	list	node number	node number	-	-
anatasks	list	node number	node number	node number	fail/unif

Appendix B: Error Messages

The following is a short overview of the error and failure messages that are issued when the compiler encounters inconsistencies and errors.

A PETFSG grammar is presupposed to contain only Prolog terms delimited by full-stops. Failures in that regard are taken care of by the Prolog system.

Grammar items (Prolog terms) which are incorrect or inconsistent are ignored, apart from triggering the system to generate an error message, e.g. the following one:

```
E R R O R: Unknown type (noune).
W A R N I N G: Form (doge) ending on line 260,
   was ignored (due to reported problem).
```

A grammar term like the following would produce this error in the context of the sample grammar, where `noune` is an unknown, i.e. undeclared, type.

```
doge >>>
  [infl_lxm,
   nfs nom_spr_agr,
   syn:[head:noune,
        spr:{[syn:[head:[det,
                         count:'+']]]},
        comps:{}],
   sem:[mode:ref,
        index: <> I,
        restr: <>[dog(I)]]].
```

A PETFSG value term (in a rule or lexical entry) that is presupposed to yield a coherent description will trigger the following kind of warning if they actually are contradictory:

```
FAILURE: 3sing contradicts non_3sing
   in "agr:non_3sing" at agr.
W A R N I N G: Form (a) ending on line 161,
   was ignored (due to reported problem).
```

This message would be produced by an incoherent lexical item entry like this one:

```
a >>>
  [word,
   syn:[head:[det,
              agr:'3sing',
              agr:'non_3sing',
              count:'+'],
        spr:{},
        comps:{}],
   sem:[mode:none,
        index: <> '-',
        restr: <>[]]].
```

The following error and failure messages may be issued by the PETFSG system during the compilation of a PETFSG grammar:

- `No current grammar defined`: No current grammar is defined, i.e. an application identifier clause is lacking.
- `Prolog variable found among grammar items`: A Prolog variable is found among the grammar items.
- `Expression with unexpected syntax (`X`)`: X is a Prolog term of a form which is not expected to occur in a PETFSG grammar.

- `Variable not allowed as an application identifier/grammar identifier/rule name/feature structure name/morphological function/suite name`: A variable is found where an atom is expected to be used as an identifier.
- `Non-atom not allowed as an application identifier/grammar identifier/rule name/feature structure name/morphological function/suite name`: A non-atom term is found where an atom is expected to be used as an identifier.
- `Additional application identifer`: An additional and superfluous application identifier is encountered.
- `Additional type declaration`: An additional type and feature declaration is found. There should be precisely one.
- `Additional initial symbol definition`: An additional initial symbol definition is found.
- `Additional leaf symbol definition`: An additional leaf symbol definition is found.
- `Type name already used (`*Type*`)`: An attempt is made to declare *Type* as the name of a type a second time.
- `Incorrect syntax in type declaration`: Incorrect syntax for the *TH* part in a declaration of the form `declaration` *TH* `where` *FL*.
- `Illegal feature declaration (`*Type*`)`: An attempt is made to associate the types `prolog_term`, `list`, or a variable with a feature declaration.
- `Features already defined for type (`*Type*`)`: An additional feature declaration for the type *Type* is found. There should be at most one.

- `"list" is a reserved type name`: An attempt is made to define a type with the name `list`, which has a reserved meaning in the system.
- `"prolog_term" is a reserved type name`: An attempt is made to define a type with the name `prolog_term`, which has a reserved meaning in the system.
- `FS name already used (`*Name*`)`: A name, *Name*, already used to name an FS, is used a second time for that purpose.
- `Prolog variable where value term was expected`: A Prolog variable is found where an expression denoting a PETFSG value was expected.
- `Unexpected kind of textual form (`*Lex*`)`: *Lex* is a non-atom appearing as the textual form of a surface word form or lexeme.
- `Unexpected variable where type was expected`: A Prolog variable is found where a type specification was expected.
- `Unexpected variable in right-hand list of rule`: A Prolog variable is found in the right-hand list of a grammar rule. Only PETFSG variables and `prolog`-terms are accepted.
- `Unexpected item in right-hand list of rule (`*A*`)`: Another kind of faulty item in right-hand list of rule is found. Only PETFSG value terms and `prolog`-terms are accepted.
- `Petfsg variable used as identifier (`*Name*`)`: A PETFSG variable is used as an identifier.
- `Unknown type (`*Type*`)`: An attempt was made to use *Type* as a type identifier, without this being supported by the type and feature declaration.

- `Unknown feature (`*Feat*`)`: An attempt was made to use *Feat* as a feature name, without this being supported by the type and feature declaration.
- `No such morphological function (`*X*`)`: *X* is occurring in a lexeme entry or lexical rule as a morphological function, while being unknown as such.
- `Inapplicable atom reshaping function (`*AR*`)`: The atom reshaping function *AR* did not give any output, because the `atom_reshaping/3` call failed.
- `Unknown fs name (`*Name*`)`: An attempt is made to access an FS by the name *Name*, which is not known to name an FS.
- `Unknown entity, cannot evaluate (`*Val*`)`: The term *Val* does not represent a valid PETFSG value term.
- `Object does not allow feature (`*Feat*`)`: The feature *Feat* is associated with an object of the wrong type.
- `Wrong value for feature (`*Feat*`)`: The feature *Feat* is assigned a value of the wrong type.
- `Failure:` *Type1* `contradicts` *Type2* `[in` *Expr*`] [at` *Path*]: Unification failure in the expression *Expr* at *Path*. The types *Type1* and *Type2* are incompatible.
- `Failure: incompatible lists [in` *Expr*`] [at` *Path*]: Unification failure at *Path* due to list incompatibility.

Appendix C: The Sample Grammar

The sample grammar (in the file `gram_sample.pl`), which has been used for illustration above, gives a toy treatment of a very small fragment of English. It is essentially an implementation of a few aspects of the Sag & Wasow (1999) grammar.

This is a full listing of the grammar file:

•

```
application('sampleII.2',
            ['English'],
            '/home/staff/matsd/petfsg/').

declaration    %% Compare Sag and Wasow (1999: 386).
    [ss_struc subsumes [          %% synsem-struc
       phrase subsumes [],
       lex_item subsumes [
         word subsumes [],
         lxm subsumes [           %% lexeme
           const_lxm subsumes [],
           infl_lxm subsumes []]]],
     agr_cat subsumes [
       '3sing' subsumes [],
       non_3sing subsumes []],
     gram_cat subsumes [],
     pos subsumes [               %% part of speech
```

```
        verb subsumes [],
        adv subsumes [],
        adj subsumes [],
        comp subsumes [],
        conj subsumes [],
        prep subsumes [],
        nom subsumes [
          noun subsumes [],
          det subsumes []]],
      sem_struc subsumes [],
      form_type subsumes [
        fin subsumes [
          pres subsumes [],
          pret subsumes []],
        inf subsumes [
          infin subsumes []]],
      mode_type subsumes [
        none subsumes [],
        prop subsumes [],
        ques subsumes [],
        dir subsumes [],
        ref subsumes []],
      boolean subsumes [
        '+' subsumes [],
        '-' subsumes []]]
where
   [ss_struc features
      [syn:gram_cat,
       sem:sem_struc],
    gram_cat features
      [head:pos,
       comps:list,
       spr:list,
```

```
       gap:list],
    pos features
      [form:form_type,
       pred:boolean],
    nom features
      [agr:agr_cat],
    det features
      [count:boolean],
    sem_struc features
      [mode:mode_type,
       index:prolog_term,
       restr:prolog_term]].

initial_symbol
   [syn:[head:[form:fin],
         spr:{},
         comps:{}]].

leaf_symbol word.

head_complement rule
   [phrase,
    syn:[head:xHead,
         comps:xComps,
         spr:xSpr],
    sem:[mode:xMode,
         index:xIndex,
         restr:xRM]]     ===>

   [[syn:[head:xHead,
          comps:xComp^xComps,
          spr:xSpr],
     sem:[mode:xMode,
```

```
           index:xIndex,
           restr:xR1]],

     [xComp,
      sem:[restr:xR2]],

     prolog [compute_semantics, xR1, xR2, xRM]].

specifier_head rule
    [phrase,
     syn:[head:xHead,
          comps:{},
          spr:{}],
     sem:[mode:xMode,
          index:xIndex,
          restr:xRM]]     ===>

    [[xSpr,
      sem:[restr:xR1]],

     [syn:[head:xHead,
           comps:{},
           spr:{xSpr}],
      sem:[mode:xMode,
           index:xIndex,
           restr:xR2]],

     prolog [compute_semantics, xR1, xR2, xRM]].

saturated_noun is_short_for
  [syn:[head:noun,
        spr:{},
        comps:{}]].
```

```
a >>>
  [word,
   syn:[head:[det,
              agr:'3sing',
              count:'+'],
        spr:{},
        comps:{}],
   sem:[mode:none,
        index: <> '-',
        restr: <>[]]].

'Kim' >>>
  [const_lxm,
   nfs saturated_noun,
   syn:[head:[agr:'3sing']],
   sem:[mode:ref,
        index: <> I,
        restr: <>[named(I,'Kim')]]].

infl_reg(identity,id).
infl_reg(sing_infl,id).
infl_reg(plur_infl,affx(s)).
infl_reg(v_infin_infl,id).
infl_reg(v_third_plural_infl,id).
infl_reg(v_third_singular_infl,affx(s)).

lexrule const_lxm_lr
  morph  identity
  input  [const_lxm,
          syn:xSyn,
          sem:xSem]
  output [word,
```

```
          syn:xSyn,
          sem:xSem].

lexrule verb_infin
  morph  v_infin_infl
  input  [infl_lxm,
          syn:[xSyn,
               head:verb],
          sem:xSem]
  output [word,
          syn:[xSyn,
               head:[form:infin]],
          sem:xSem].

lexrule verb_pres_3sing
  morph  v_third_singular_infl
  input  [infl_lxm,
          syn:[xSyn,
               head:verb],
          sem:xSem]
  output [word,
          syn:[xSyn,
               head:[form:pres],
               spr:{[syn:[head:[agr:'3sing']]]}],
          sem:xSem].

walk >>>
  [infl_lxm,
   syn:[head:verb,
        spr:{[nfs saturated_noun,
              sem:[index: <>I]]},
        comps:{}],
   sem:[mode:prop,
```

```
        index: <> S,
        restr: <>[walk(S,I)]]].

love >>>
  [infl_lxm,
   syn:[head:verb,
        spr:{[nfs saturated_noun,
              sem:[index: <>I]]},
        comps:{[nfs saturated_noun,
                sem:[index: <>J]]}],
   sem:[mode:prop,
        index: <> S,
        restr: <>[love(S,I,J)]]].

nom_spr_agr is_short_for
  [syn:[head:[agr:xAgr],
        spr:{[syn:[head:[agr:xAgr]]]}]].

lexrule '3sing_noun'
  morph  sing_infl
  input  [infl_lxm,
          syn:[xSyn,
               head:noun],
          sem:xSem]
  output [word,
          syn:[xSyn,
               head:[agr:'3sing']],
          sem:xSem].

lexrule non_3sing_noun
morph  plur_infl
  input  [infl_lxm,
          syn:[xSyn,
```

```
                head:noun],
          sem:xSem]
  output [word,
          syn:[xSyn,
                head:[agr:non_3sing]],
          sem:xSem].

dog >>>
  [infl_lxm,
   nfs nom_spr_agr,
   syn:[head:noun,
        spr:{[syn:[head:[det,
                         count:'+']]]},
        comps:{}],
   sem:[mode:ref,
        index: <> I,
        restr: <>[dog(I)]]].

child >>>
  [infl_lxm,
   nfs nom_spr_agr,
   syn:[head:noun,
        spr:{[syn:[head:[det,
                         count:'+']]]},
        comps:{}],
   sem:[mode:ref,
        index: <> I,
        restr: <>[child(I)]]].

suppletive_form(child,plur_infl,children).

postparse_procedures([sem_only]).
```

```
start_page_section([
   '<p>PETFSG sample grammar. See document',
   'ation <a href="http://stp.ling.uu.se/',
   '~matsd/petfsg/II.2/">here</a>.</p>']).

css_url(
'http://stp.ling.uu.se/~matsd/petfsg/II.2/css/pgt.css').

type_color(ss_struc,'#98FB98').
type_color(phrase,'#98FB98').
type_color(lxm,'#FA8072').
type_color(const_lxm,'#FA8072').
type_color(infl_lxm,'#FA8072').
type_color(word,'#98FB98').
type_color(gram_cat,'#F0FFFF').
type_color(sem_struc,'#FFFFCC').
type_color(pos,'#7FFF00').
type_color(nom,'#7FFFD4').
type_color(noun,'#7FFFD4').
type_color(perspron,'#7FFFD4').
type_color(adj,'#CCFF00').
type_color(adv,'#CCFF00').
type_color(verb,'#FFFFCC').
type_color(v,'#FFFFCC').
type_color(cplzer,'#FFC0CB').
type_color(mod_sort,'#E6E6FA').
type_color(tds_type,'#F5FFFA').

suite(s1,[[act=types],
          [act=word, str='walks'],
          [act=parse, str=['Kim', walks],
           a1=nomod, a2=no],
          [act=parse, str=['Kim', loves, a, dog],
```

```
           a1=sem_only, a2=no],
          [act=parse, str=[a, dog, walks],
           a1=nomod, a2=yes]]).

suite(s2,[[act=lifor],
          [act=lilex],
          [act=word, str='dog'],
          [act=rules],
          [act=lexrules],
          [act=nfs],
          [act=parse, str=['Kim', loves, a, dog],
           a1=nomod, a2=no],
          [act=parse, str=['Kim', loves, a, dog],
           a1=nomod, a2=yes],
          [act=active, str=['Kim', loves, a, dog],
           a1=0, a2=1],
          [act=anatasks, str=['Kim', loves, a, dog],
           a1=1, a2=2, a3=3, a4=fail]]).
```

•

Appendix D: Prolog Procedures

This is a listing of the file `application_procedures.pl`. It contains the Prolog procedures required by the sample grammar. These are documented in Chapter 3. Comments on the code appear in boxes.

•

```
:- use_module(library(lists)).

:- use_module(library(fs_handling),
              [path_value/3]).
```

label_token/2
Tokens are not labeled.

```
label_token(_,_,_).
```

postparse_procedure/3
This postparse procedure clause selects only the `sem` feature value and incorporates it in a new `ss_struc` object.

```
postparse_procedure(sem_only,ALL,[SEL]):-
    !,
    path_value(ALL,sem,SEM),
    path_value(SEL,sem,SEM).
```

call_prolog/1
This procedure takes care of Prolog calls made from inside the phrase structure rules.

```
call_prolog([compute_semantics,
             Semantics1,
             Semantics2,
             Semantics3]):-
         compose_semantics(Semantics1,
                           Semantics2,
                           Semantics3).
```

compute_semantics_procedure/3
This procedure computes a semantic value by means of appending restriction lists.

```
compose_semantics(prolog_term(Semantics1,_),
                  prolog_term(Semantics2,_),
                  prolog_term(Semantics3,_)):-
    append(Semantics1,
           Semantics2,
           Semantics3).
```

atom_reshaping/3
`id` is the identity atom reshaping function. The `affx(`*Affix*`)` atom reshaping function adds an affix (*Affix*) to a word form atom.

```
atom_reshaping(id,Atom,Atom).

atom_reshaping(affx(Affix),Atom_in,Atom_out):-
    !,
    name(Atom_in,Atom_in_name),
    name(Affix,Affix_name),
    append(Atom_in_name,
```

```
            Affix_name,
            Atom_out_name),
    name(Atom_out,Atom_out_name).
```

dsp_prolog_term/4

This procedure displays Prolog terms embedded within FSs.

```
dsp_prolog_term(Term,
                _,
                Variable_number_in,
                Variable_number_out):-
    instantiate_variables(Term,
                          Variable_number_in,
                          Variable_number_out),
    write(Term).
```

instantiate_variables/3

The instantiate_variables/3 call gives us a new numbering of the variables. Atoms simply replace the variables. The first argument is the term to be modified in this way. The second and third arguments are the numbers for the next variable number before and after the term has been instantiated.

```
instantiate_variables(Term,
                      Variable_number_in,
                      Variable_number_out):-
    var(Term),
    !,
    make_atomic_symbol_for_variable(
                        Term,
                        Variable_number_in,
                        Variable_number_out).

instantiate_variables(Term,
```

```
                        Variable_number,
                        Variable_number):-
    simple(Term),
    !.

instantiate_variables([Head|Tail],
                        Variable_number_in,
                        Variable_number_out):-
    !,
    instantiate_variables(Head,
                            Variable_number_in,
                            Variable_number_aux),
    instantiate_variables(Tail,
                            Variable_number_aux,
                            Variable_number_out).

instantiate_variables(Term,
                        Variable_number_in,
                        Variable_number_out):-
    !,
    Term =.. List,
    instantiate_variables(List,
                            Variable_number_in,
                            Variable_number_out).
```

`make_atomic_symbol_for_variable/3`
A call `make_atomic_symbol_for_variable(-`*Symbol*`,+`*Number_in*`,-`*Number_out*`)` creates a term for a variable in the style of e.g. '`<u>P5</u>`', if 5 is the `Number_in`. The third argument is the second argument counter increased by 1.

```
make_atomic_symbol_for_variable(Symbol,
                                Number_in,
                                Number_out):-
```

```
name(Number_in,
     Number_name),
append([60,117,62,80|Number_name],
       [60,47,117,62],
       Symbol_name),
name(Symbol,Symbol_name),
Number_out is Number_in + 1.
```

•

Appendix E: Internal Form of PETFSG Values

These are the basic ideas behind the internal Prolog representation of PETFSG values: The operator `<&>/2` connects terms representing types with sequences of feature values. These sequences are `fttrm`-terms. A `fttrm`-term of arity $n > 0$ is a compound term with `fttrm`/n as its functor and n arguments. The `fttrm`-term of arity 0 is the atom `fttrm`.

Let us indicate the specificity of each user-defined PETFSG type with the help of an integer. The most general types are associated with the number 0, and if a type is associated with the specificity number n, each one of its immediate subtypes is associated with $n+1$.

An internal level n PETFSG value term of type *Type* is of one of the following kinds:

- A term of the form *Type*`(`VT_{n+1}`)<&>`*Fttrm*, such that the functor *Type* is of arity 1, *Type* is typographically identical to a user-defined type of specificity n, the user-defined type called *Type* has at least one subtype and is associated with m features, and *Fttrm* is a `fttrm`-term of arity m, and VT_{n+1} is an internal level $n+1$ PETFSG value term of type *SubType*, which is typographically identical to an immediate subtype to the user-defined type called *Type*.

- A term of the form *Type*`<&>`*Fttrm*, such that *Type* is an atom, *Type* is typographically identical to a user defined type of specificity n, the user-defined type called *Type* does not have any subtype, and

the user-defined type called *Type* is associated with *m* features, and *Fttrm* is a `fttrm`-term of arity *m*.

The `fttrm`-terms carry the feature values embedded within an internal level *n* PETFSG value term. Each argument of a `fttrm`-term is a PETFSG value term.

An absolute internal PETFSG value term is of one of these kinds:

- An internal level 0 PETFSG value term.
- `list_type(list_type_e#fttrm)` which is the internal form of an instance of the PETFSG empty list.
- Of the form `list_type(list_type_ne#f(`*Head*`,`*Tail*`))`, where *Head* is the list head and *Tail* is the list tail. *Head* is an absolute internal PETFSG value term and *Tail* is a PETFSG internal list.
- Of the form `prolog_term(`*Term*`,`*Var*`)`, where *Term* is any prolog term and *Var* is a variable (in order to preserve information about structure sharing). Such a term is used to represent a PETFSG prolog term value.

For instance, the structure written in this way:

```
[syn:[head:[agr:xAgr],
      spr:{[syn:[head:[agr:xAgr]]]}]]
```

receives this internal representation, given the sample grammar:

```
ss_struc(_988)<&>fttrm(
  gram_cat<&>fttrm(                                  %% syn
    pos(                                             %% head
      nom(_1405)<&>fttrm(
        agr_cat(_1398)<&>fttrm))<&>_1192,            %% agr
    _1195,                                           %% comps
```

```
   list_type(                                    %% spr
     list_type_ne<&>fttrm(
       ss_struc(_1969)<&>fttrm(                  %% list hd
         gram_cat<&>fttrm(                       %% syn
           pos(                                  %% head
             nom(_2386)<&>fttrm(
               agr_cat(_1398)<&>fttrm))<&>_2173,
                                                 %% agr
         _2176,                                  %% comps
         _2177,                                  %% spr
         _2178),                                 %% gap
       _1967),                                   %% sem
     list_type(                                  %% list tl
       list_type_e<&>fttrm))),
   _1197),                                       %% gap
  _986)                                          %% sem
```

Appendix F: PETFSG-II Source Files

The downloadable PETFSG files are found under the URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/pgt/`.

The following Prolog modules, each one defined in a file whose name is the module name with the addition of `.pl`, are involved in the PETFSG-II system:

`compiler`**:** This module contains the procedures responsible for compiling grammar files and for creating a PETFSG application from them. It also contains the procedure for creating the CGI script.

`main_petfsg`**:** This module contains the central procedures of the grammar tool, such as the main control mechanisms, the parser, the task analyzer, and the facilities for displaying grammar items.

`grammar`**:** This module contains the dynamic predicates in which the compiled grammar is stored. It also contains a few procedures for accessing these predicates.

`fs_handling`**:** This module contains procedures which are used to create the internal Prolog term representation of feature structures. It also contains the dynamic predicates encoding the types and features of a particular grammar.

`fs_display`**:** This module contains procedures for displaying feature structures. Both `tab` and `txt` styles, as described in Section 4.4, are handled here.

`chart`: This module contains the dynamic predicates in which the edges of the chart are stored. It also contains a few procedures for accessing these predicates.

`html`: This module contains the procedures used for creating HTML output.

`database`: This module is used to store various interface-related data of global significance.

`cgi_parameters`: This module contains predicates which access the key-value list associated with the URI request sent to the PETFSG CGI script.

`this_prolog`: This module contains procedures whose definitions depend on whether SICSTUS or SWI Prolog is used. It should be identical to either `sicstus.pl` or `swi.pl`.

`this_application`: This module contains Prolog procedures which are associated with the individual grammar of the application, rather than with the PETFSG-II system as documented here. The file `this_application.pl` simply requires that the file `application_procedures.pl` be compiled.

The following Prolog files are associated with the previous modules, as mentioned above:

`sicstus.pl`: This file is to be copied onto `this_prolog.pl` when SICSTUS Prolog is used.

`swi.pl`: This file is to be copied onto `this_prolog.pl` when SWI Prolog is used.

Futhermore, the following two files are useful:

`start_up_petfsg.pl`: Contains start-up SICStus Prolog procedures. Used to create a general SICStus start-up saved state.

`operators.pl`: The collected operator declarations (which come from the `compiler` and `fs_handling` modules). This file is needed only when SWI Prolog is used.

`pgt.css`: The CSS used here.

The sample grammar application is based on the following files, which are found under the URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/sample/`:

`gram_sample.pl`: The sample grammar.

`application_procedures.pl` (also mentioned above): This file contains SICSTUS Prolog procedures used by the sample grammar, to be loaded into the module `this_application`, as required in the file `this_application.pl`. Its content id appropriate for the sample grammar application. It would probably be modified for a different application.

`make_sample.pl`: This file provides a convenient way of creating the saved state from the sample grammar. It defines the necessary steps under SICSTUS Prolog.

`make_sample_swi.pl`: The SWI Prolog version of the previous file.

Report 2

An Implementation of Token Dependency Semantics for a Fragment of English

Preface

This report describes an implementation of a computational semantics for a fragment of English. Its main purpose is to illustrate the application of a semantic framework which is called 'Token Dependency Semantics'. The basic principles are outlined in my article 'Token Dependency Semantics and the Paratactic Analysis of Intensional Constructions' (published in the *Journal of Semantics*, 2002). The present implementation involves an HPSG-style formal grammar and it is based on the system and formalism 'Prolog-embedding typed feature structure grammar' version II.2 (PETFSG-II, see the first part of the present volume or URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/` for documentation). A report on a previous and in several respects different implementation of this kind of semantics was given in *RUUL* (Reports from Uppsala University Dept of Linguistics) **34**, 1999. A demo, updates on this system, source files, and related material are found at URL `http://stp.ling.uu.se/~matsd/tds/jos/`.

I want to thank two anonymous referees of the *Journal of Semantics* and its editor Peter Bosch for valuable comments on earlier versions of this semantic framework. I am also grateful to Bengt Dahlqvist, Roussanka Loukanova, Leif-Jöran Olsson, Per Starbäck, Anna Sågvall Hein, Åke Viberg, and Per Weijnitz for help and suggestions.

Uppsala, December 2002

Mats Dahllöf (`mats.dahllof@ling.uu.se`)

1 Introduction

The purpose of this report is to document the technical details of an implementation of 'Token Dependency Semantics' (henceforth TDS). The implementation is based on an HPSG-style formal grammar, in which the semantics is embedded. The theoretical background and motivation for this approach to computational semantics is given in my article 'Token Dependency Semantics and the paratactic analysis of intensional constructions' (Dahllöf 2002). The grammar as implemented here uses the system and formalism 'Prolog-embedding typed feature structure grammar' version II.2 (PETFSG-II, see the first part of the present volume or URL `http://stp.ling.uu.se/~matsd/petfsg/II.2/` for documentation). This report presupposes familiarity with both the theoretical article and the PETFSG-II system. The relation between the present report and the background article is fairly direct, but there are a few discrepancies which will be noted in the course of the presentation.

The main purpose of this document is thus to give the reader some help in understanding the implemented TDS grammar and the auxiliary Prolog procedures. They are listed, along with comments and explanations, in the appendices. The grammar mainly provides treatments of a number of intensional constructions, and it covers only simple examples of other kinds of phrase. The present grammar also proposes treatments of a few phenomena not mentioned in the background article, e.g. subject raising, object raising and object control verbs, auxiliary verbs, and noun-modifying adjectives.

The PETFSG-II system uses typed feature structures in which

Prolog-terms may be embedded.[1] The grammar associates sentences with semantic representations which (in many cases) are underspecified with respect to scopal relations. A few additional Prolog procedures, external to the grammar, are provided in this PETFSG application. These are applied as 'postparse operations' and are used to compute the possible scopal resolutions. This provides a way of testing the underspecified semantic representations against our intuitions about possible readings (and ambiguity).

A demo and the files of the implementation are available at the URL `http://stp.ling.uu.se/~matsd/tds/jos/`. A number of test suites are provided for those who want to see the features of the grammar illustrated in a comprehensive way. The names of the source files of the application are listed in Appendix E, along with brief comments on their content.

1.1 The `sign` Type and its Features

The type `sign` is a subtype of the type `aops`, to be understood as *absent or present sign*. The only other subtype is `neg_s`, which is used to negate the presence of a sign. (This possibility is used to prevent expressions from being associated with heads as a modifiers. See Section 2.1.)

A `sign` FS is associated with features as follows:

```
sign features
  [token:prolog_term,
   head:head_sort,
   spr:list,
   comps:list,
   slash:list,
```

[1] Prolog procedures may be called from inside PETFSG rules, but the present grammar, which relies only on unification, does not make use of this possibility.

```
   cont:cont_sort,
   pm:punct_mark]
```

The features are used as described in the background article, with the exception of `slash` and `pm`. The `slash` feature is not yet put to use. The value of the feature `pm` is the punctuation mark delimiting a phrase, or a value indicating the absence of a punctuation mark. (Consult the grammar code for details.)

1.2 The `cont`(ent) Features

The compositional system is formulated according to the 'hooks-and-holes' model of Copestake et al. (2001). The semantic information structures and operations are thereby disentangled from the syntactic ones. Each semantic representation contains a 'hook' and a number of labelled 'holes'. The holes are 'gaps' in the semantic structure. So, a constituent 'fills' a hole of a sister constituent, when the hook of the first constituent is unified with this hole.

The value of the `sign` feature `cont`(ent) has a value of the type `cont_sort`.

```
cont_sort features
   [hook:hole_type,
    h:holes,
    tds:prolog_term,
    qeqs:prolog_term,
    tree:prolog_term,
    rss:prolog_term],
```

The feature `hook` carries the semantic 'hook'. The `tds` (token dependency statements) feature carries the basic semantic information concerning immediate outscoping and coindexation. The `qeqs` feature carries the `qeq` (equality modulo quantifiers) statements. The `tds`

and `qeqs` values are difference list pairs, joined by a minus ('`-`') infix functor. (This makes it possible to collect this information without the use of an append operation.) In Dahllöf (2002) the items of the present `tds` and `qeqs` lists are collected in one set (list), under the feature TDS.

The 'hole' features are collected under the feature `h` (see below). The features `tree` and `rss` (reading-specific statements) are used in the computation of scopally resolved readings (see Section 3.2). They are left unspecified by the semantics as defined by the grammar.

The values of the `hook` feature and the hole features are of the type `hole_type`, defined as follows by relevant parts of the type and feature declaration:

```
hole_type subsumes [
   closed subsumes [],
   hole subsumes [
     hole_n subsumes [],
     hole_v subsumes []]],

hole features
   [ltop:prolog_term],

hole_n features
   [key:prolog_term],

hole_v features
   [extarg:prolog_term],
```

The non-`closed` holes are of two kinds: nominal (`hole_n`) ones, carrying `ltop` and `key` features, and verbal (`hole_v`) ones, carrying `ltop` and `extarg` features.

The feature `h` carries a `holes` value, as follows:

```
holes subsumes [
```

```
     holes_v subsumes [],
     holes_d subsumes []],

holes features
   [mod_hole:hole_type],

holes_v features
   [subj_hole:hole_type,
    comp_holes:list],

holes_d features
   [spec_hole:hole_type]
```

So, the hole collections (of the `holes` type) are either verbal (`holes_v`) or determiner-related (`holes_d`). The `holes_v` hole collections comprise a `mod_hole`, a `subj_hole`, and the complement holes, collected under `comp_holes` (see below). The `holes_d` structures carry a `mod_hole` and a `spec_hole`.

The treatment of complement holes is different from that of Dahllöf (2002) and Copestake et al. (2001). Here, the `comp_holes` feature carries a list value. Its elements are `hole` objects, each associated with the corresponding element on the `comps` list. This is a matter of technical convenience, and is not intended to reflect a theoretically significant decision.

1.3 The Representation of TDS Information

The TDS relations are represented by the following symbols (with their appearance in Dahllöf (2002) in parentheses):

`c1` **(‘c_1’):** Coindexation between a quantifier token and a predicate token with respect to the first argument position.

`c2` **(‘c_2’):** Coindexation between a quantifier token and a predicate token with respect to the second argument position.

`restr` **(‘D_{restr}’):** Immediate outscoping between a quantifier token and its restriction.

`body` **(‘D_{body}’):** Immediate outscoping between a quantifier token and its body.

`ptop` **(‘$D_{\text{p-top}}$’):** Immediate outscoping between a (paratactic predicate) token and the scopally topmost node of its paratactic argument.

`qeq` **(‘$=_q$’):** Equality modulo quantifiers.

`lx` **(‘`L`’):** Holds of a token, the lexeme or word form of which it is an instance, and a semantic valency label (see below).

`parg` **(‘$A_{\text{p-arg}}$’):** Holds of a (paratactic relation) token and its paratactic argument subtree.

`anchor` **(‘A_{anchor}’):** Holds of a (paratactic relation) token and its anchor argument.

`Ind` **(‘Ind’):** Is used in such a way that ‘$\mathsf{Ind}(t, n)$’ is the individual associated with argument position n of the token t given current bindings.

`+` **(‘+’):** Token aggregation function.

`v_sf:` Is only used in the implemented grammar. See Section 2.2.

The `lx/3` terms give information about *semantic valency*. This information is used by the scopal resolution procedure. These valencies are represented by atomic identifiers as follows:

`q1:` One-place quantifier (i.e. a name).

`q2`**:** Two-place quantifier (i.e. *two*).

`p_a1`**:** Intransitive non-paratactic predicate (i.e. *smile*).

`p_a1_a2`**:** Transitive non-paratactic predicate (i.e. *read*).

`p_para`**:** Intransitive paratactic predicate (i.e. *probably*).

`p_a1_para`**:** Transitive predicate whose second argument is a paratactic one (i.e. *say*, *intentionally*).

`p_a1_a2_para`**:** Bitransitive predicate whose third argument is a paratactic one (i.e. *persuade*).

2 Notes on the Grammar

In this section I will turn to the implemented grammar and to some aspects of it which are not documented in the main article on TDS (Dahllöf 2002). They relate to the phrase structure rules, to tense, and to a number of additional constructions, e.g. prepositional objects, raising and control phenomena, and adjectives modifying nouns.

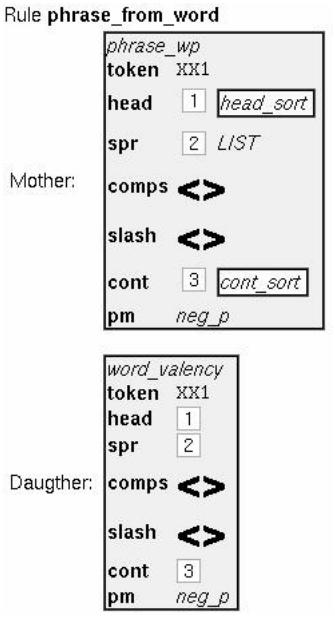
2.1 Phrase Structure Rules

A few comments should be made concerning the phrase structure rules. The 'official' HPSG head-complement rule is here implemented by means of two rules. One of them is for the case in which no complement is present. The other one allows a head to find a complement sister, one at a time. Due to semantics, the specifier-head construction gives us two rules. One is for the subject case, in which the (verbal) head is also the semantic head. The other rule is for the determiner specifier case, in which a determiner, which is a semantic head, is associated with a syntactic (nominal) head. In order to make it easy to find out which rule has produced a phrase, there is a subtype to the type `phrase` for each phrase structure rule. These distinctions do not play any linguistic role, as the lexical items or the phrase structure rules do not contain any FSs at that level of specificity. These are the `phrase` subtypes:

`phrase_wp`	phrase from word construction
`phrase_hc`	head-complement construction
`phrase_ssh`	subject specifier-head construction
`phrase_sph`	determiner specifier-head construction
`phrase_mh`	modifier-head construction
`phrase_hp`	head-punctuation-mark construction

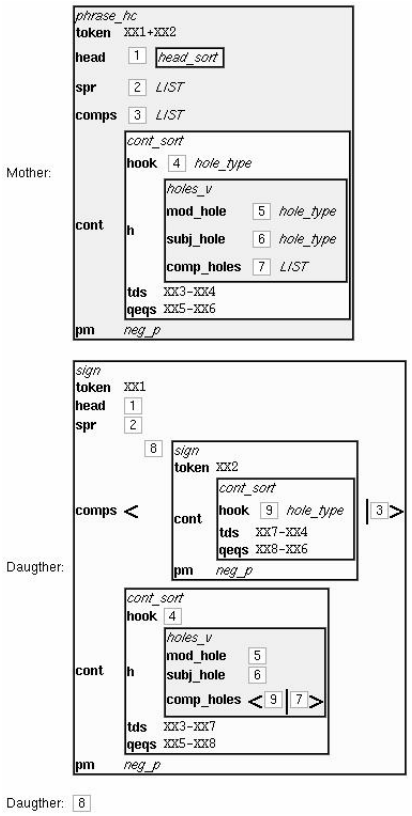
In order to eliminate the need for unnecessary rule applications, the valency-taking surface word forms have been assigned the type `word_valency`. The type `word_no_valency` is for words which belong to classes which never, at least as far as the present fragment of English is concerned, take complements or specifiers. These words are never selected as heads by the head-complement rules. They are not selected as heads by the specifier-head rules either, as it is the head-complement rules that turn words into phrases. In the phrase structure rules, the mother's `cont:tds` value is given by appending the `tds` values on the daughters. This is done by means of difference lists.

The grammar contains six phrase structure rules. Let us first take a look at the rule `phrase_from_word`:



This rule covers the case in which the head-complement rule applies with an empty list of complements, i.e. when the mother `phrase` dominates a head `word` without sisters.

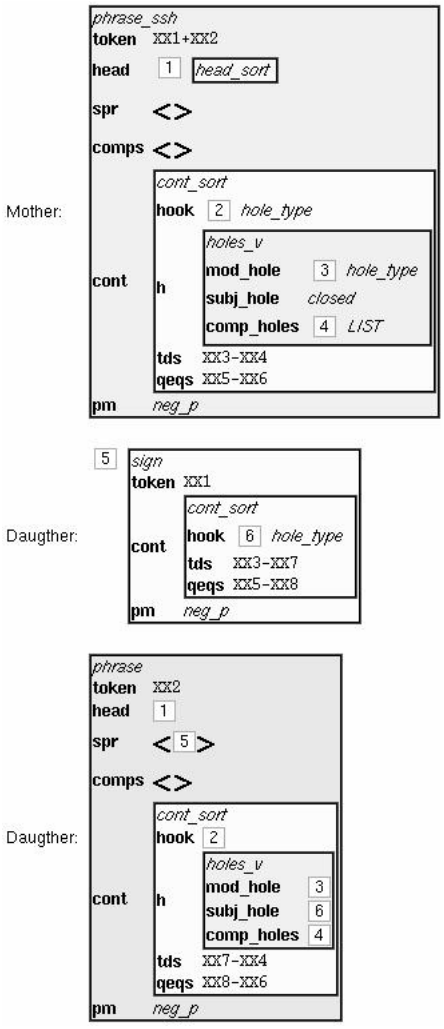
The `hd_comps` rule covers the case in which the head-complement rule applies with a non-empty `comps` list on the head.



Only one complement is incorporated on each application of the rule. The complement holes in the `comp_holes` list correspond one-by-one to the `comps` list items. So, the values of the two features on the mother are equal to the tails of the corresponding values on the head daughter.

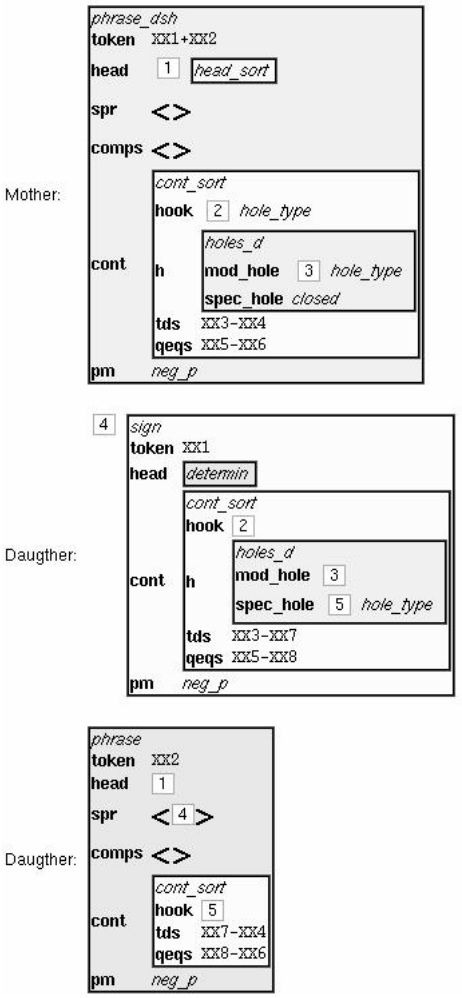
The `subj_spr_hd` rule is for specifier-head constructions in which the specifier is a subject to a verbal (syntactic) head, which is also the

semantic head.[1]

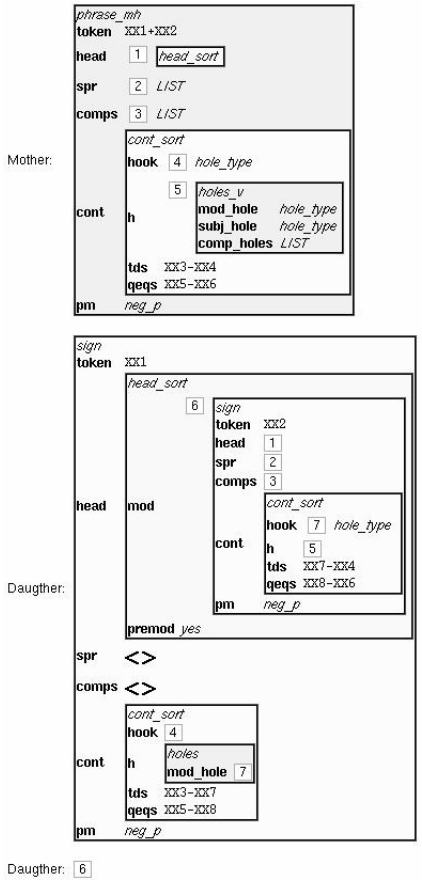


[1]Pollard's & Sag's (1994: 51) 'Spec Principle' is not put to use in this grammar. The kind of circularity it involves cannot be implemented in a PETFSG grammar.

The `det_spr_hd` rule is for specifier-head constructions in which the specifier is a determiner and the syntactic head a noun. The determiner is the semantic head here.

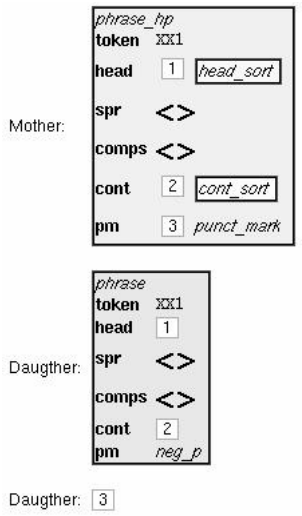


The premodifier-head rule is called `mod_hd`, and it looks like this:



The `head:premod` value (boolean) on the modifier defines it as a premodifier, while the `head:mod` value describes the modified constituent.

Finally, there is the phrase-punctuation-mark rule `phr_punct`:



A phrase without a punctuation mark—formally, `pm:neg_p`—combines with a punctuation mark and forms a phrase with the feature `pm` specifying the sister punctuation mark. The mother and daughter phrases otherwise carry the same feature values.

2.2 Tense, Perfectivity, and Progressivity

The grammar only 'keeps track' of tense, perfectivity, and progressivity. A seriously worked-out proposal would probably be quite different, for reasons that are mentioned below.

The tense, perfectivity, and progressivity information is associated with a verbal `head` feature `v_sf` (verbal semantic features). This value is a complex term [*T*,*P*,*Q*] containing three items. *T* is for basic tense. There are three possible values for *T*: present (-), preterit (+), and tenseless (n, for the case of infinitive). The *P* value represents perfectivity, non-perfect (-) and perfect (+) being the two alternatives. Finally, *Q* stands for progressivity. Non-progressive (-) and progressive (+) are the two possibilities here. Examples:

[−,−,−]	pres, non-perf, non-progr	*Mary smiles.*
[−,−,+]	pres, non-perf, progr	*Mary is smiling.*
[−,+,−]	pres, perf, non-progr	*Mary has smiled.*
[−,+,+]	pres, perf, progr	*Mary has been smiling.*
[+,−,−]	pret, non_perf, non-progr	*Mary smiled.*
[+,−,+]	pret, non-perf, progr	*Mary was smiling.*
[+,+,−]	pret, perf, non-progr	*Mary had smiled.*
[+,+,+]	pret, perf, progr	*Mary had been smiling.*

In a lexical token node, the tense, perfectivity, and progressivity value is also associated with an entry on the semantic `tds` list. These entries are of the form `v_sf(`*Token*`,[`*T*`,`*P*`,`*Q*`])`, where *Token* is a token identifier and *T*, *P*, and *Q* are as above.

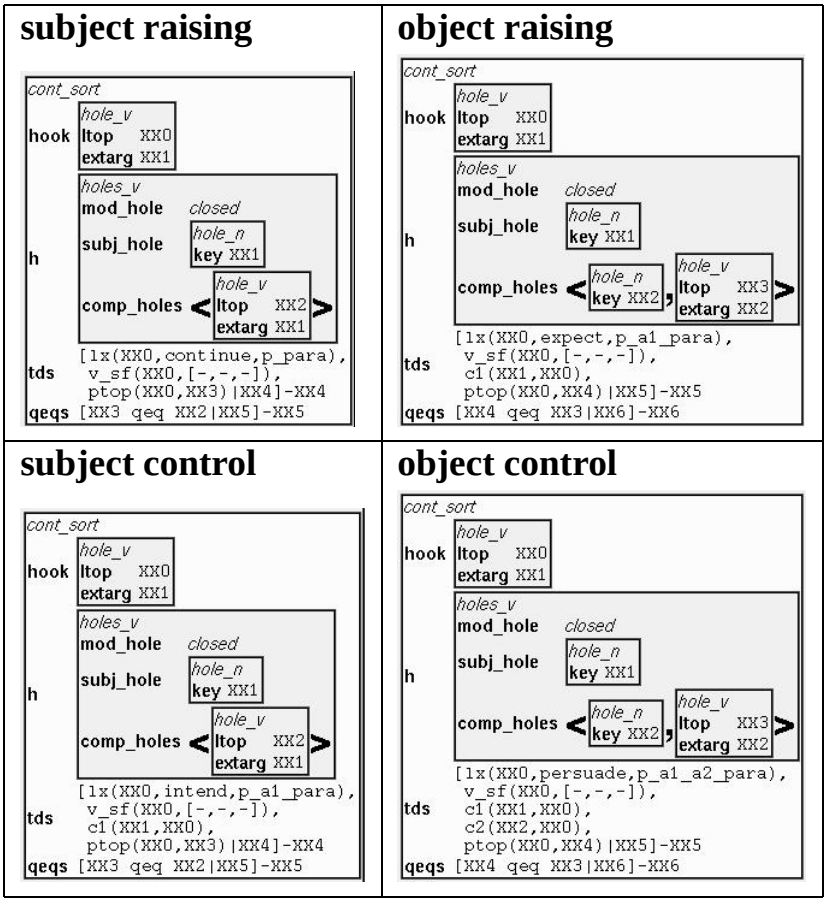
An auxiliary verb taking an appropriate kind of VP as its complement is allowed to constrain the complement's `head:v_sf` value. In this way, all of the tense, perfectivity, and progressivity information will ultimately be tied to the main verb form token.

This analysis takes tense, perfectivity, and progressivity to be a matter of properties of main verb tokens. However, it seems that e.g. tense may enter the picture as a scope-taking operator, instances of which often enter into underspecified scopal relations with, say, NP's. It is thus be motivated to think of 'tense tokens' as forming their own scopal nodes. Still, the analysis in terms of binary features may be of some use.

2.3 Control and Raising Verbs

In raising and control constructions, an argument position (corresponding to the subject) on the head verb predicate of a complement infinitival phrase is bound from the outside via the `extarg` feature, as shared by the complement hole and the complement's hook. In subject control and raising, this `extarg` value derives from the subject

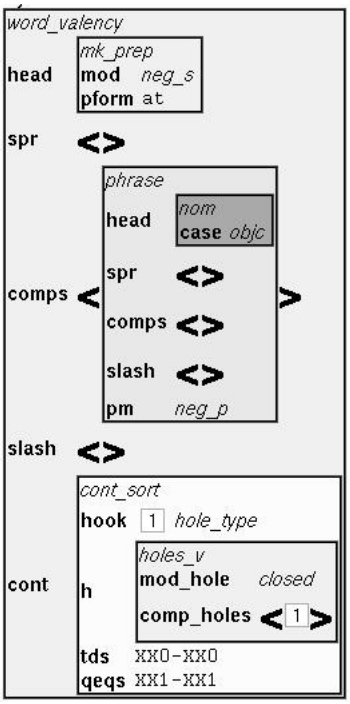
of the verb taking the complement, while an object is the source in object control and raising. The difference between control and raising is that the governing verb relation carries an argument position for the `extarg`-supplying quantifier in control cases, while it does not in raising cases. This systematicity is easily seen in the following `cont`(ent) values from instances of the four kinds of verb.



2.4 Prepositional Objects

Prepositional objects are complement PPs in which the preposition does not stand for a predicate. Prepositions of this kind are assigned to their own class, `mk_prep` (marker preposition, cf. Sag & Wasow 1999). A PP formed by a `mk_prep` head and its complement NP has a

`cont`(ent) which identitical to that of the complement NP. The form of the preposition is captured by a feature `head:pform`. So, such a preposition can be defined as follows, *at* being the example here:



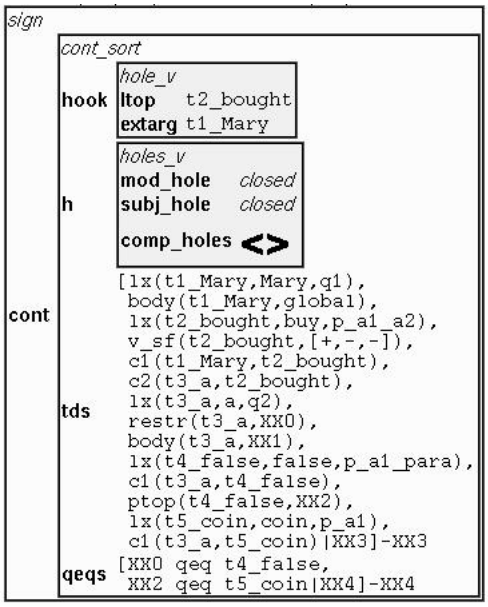
Now, a `mk_prep` PP complement is treated just as an ordinary NP object as far as semantics is concerned.

2.5 Noun-Modifying Adjectives

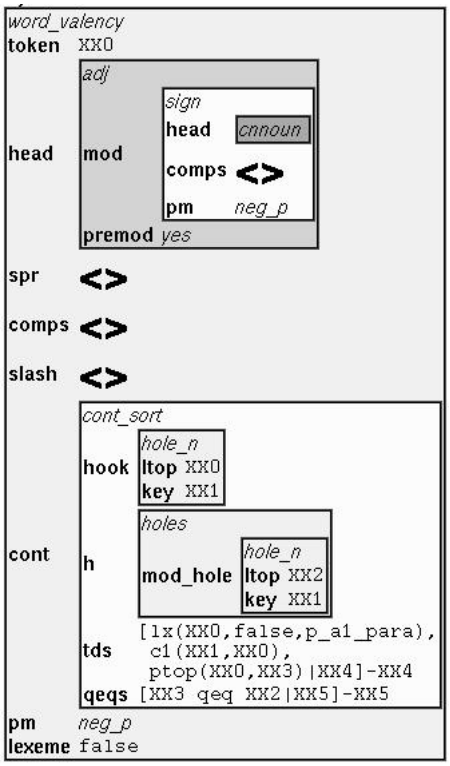
The grammar gives us a paratactic analysis of adjectives in attributive (modifier) position. This is only a tentative proposal and the present remarks do not exhaust the issue of adjective semantics.

An adjective in attributive position may be analyzed as two-place relations holding of an entity and a description provided by the modified noun predication token. Thereby, we make explicit the way in which an adjective predication may be understood as being dependent

on the noun predication. For instance, a false coin may be a genuine piece of metal. So, *false* in *false coin* may be understood as a relation between an object and a description, characterizing as what the object is considered. Under the present paratactic mode of analysis, this description is most naturally identified with the relevant noun (complex) token. This idea gives ut the following analysis for the sentence *Mary bought a false coin.*



Now, the adjective may be defined as follows:



The adjective's first argument will be bound by the same quantifier as the noun's first argument, as provided by the `cont:hook:key` value. The adjective also takes a paratactic argument, whose scopally topmost token is `qeq` the local top of the modified noun complex.

When it comes to adjectives that can be understod as standing for one-place predicates (properties), we may assume that the paratactic argument simply is required to be true. So, if *red* in *red book* is understood in that way, *red* is true of an 'absolutely' red object and another true description of that object.

A problem with this analysis is that the 'considered-as' connection between the adjective and the noun seems to be a matter of pragmatic probability rather than absolute semantics. For instance, something that is described as a *false painting* may be a genuine painting that has been painted with the intention of making people think it is a van Gogh. So, the *false painting* may be false only considered as a van Gogh painting, not considered as a painting.

It is beyond the scope of the present report to explore these issues more deeply. My present intention is rather to show that the TDS framework provides the formal means of stating this kind of solution.

3 Prolog Procedures

The TDS application makes use of general Prolog procedures for two purposes: First, there is a need for a mechanism to generate unique identifiers for the tokens of an input string. Secondly, the TDS implementation involves a few postparse procedures. These make partial selections of information to be displayed and/or compute scopal resolutions from underspecified semantic representations. The following ones are defined:

- `nomod`: No postparse operation is applied.
- `cont`: Only the value of the feature `cont` is selected for output.
- `scopal_resolution`: The set of scopal resolutions of the `cont` value are computed. Only the value of the feature `cont` is then selected for output.

3.1 Token Identifier Assignment

The TDS grammar presupposes that the grammatical descriptions uniquely identify the tokens to which they apply. This means that each token must be assigned a unique identifier. This is done procedurally with the help of the predicate `label_token/3`, which is called just before a lexical inactive edge is stored in the chart:

```
label_token(Description,Number,Lexical_atom):-
    name(Number,Number_name),
```

```
name(Lexical_atom,Lexical_atom_name),
append([116|Number_name],
       [95|Lexical_atom_name],
       New_name),
name(Token_identifier,New_name),
path_value(Description,token,
       prolog_term(Token_identifier,_)).
```

The `label_token/3` arguments are the FS and the number giving the position of the word in the input string (an integer). The token identifier, assigned to the feature `token`, is an atom formed by `t` (ascii 116) followed by the position number, an underscore character (ascii 95), and the ortographic form. For instance, `t3_house` is the result, if the token is an instance of *house* at position 3.

3.2 Specification of Readings

The `cont`(ent) features `rss`, and `tree` are used in the specification of scopally resolved readings. The `rss` (reading-specific statements) feature carries a list of additional TDS statements pertaining to the reading in question. The `tree` feature, which is redundant given the other features, gives us an easy to read picture of the TDS tree representing the reading. (The predicate `dsp_prolog_term/4` is responsible for displaying these tree structures in a suitable fashion, as described in the PETFSG-II documentation.)

The specification of scopally resolved readings is best explained with reference to an example. (This example also appears in Appendix A.) The sentence *Mary says that two students smiled* receives the following underspecified TDS representation, the immediate outscoping statements (having `restr/2`, `body/2`, or `ptop/2` for main functor) and coindexation statements (having `c1/2` or `c2/2` for main functor) being collected under the `tds` feature and the `qeq` statements under

the `qeqs` feature. The two lists are actually implemented as difference lists but this is not shown here.

```
[lx(t1_Mary,Mary,q1),
 body(t1_Mary,global),
 lx(t2_says,say,p_a1_para),
 v_sf(t2_says,[-,-,-]),
 c1(t1_Mary,t2_says),
 ptop(t2_says,XX0),
 lx(t4_two,two,q2),
 restr(t4_two,XX1),
 body(t4_two,XX2),
 lx(t5_students,student,p_a1),
 c1(t4_two,t5_students),
 lx(t6_smiled,smile,p_a1),
 v_sf(t6_smiled,[+,-,-]),
 c1(t4_two,t6_smiled)]

[XX0 qeq t6_smiled,
 XX1 qeq t5_students]
```

Here, underspecification is due to the Prolog variables `XX0`, `XX1` and `XX2`. The scope resolution algorithms work by instantiating variables.

One of the (two) readings of this example given by the `scopal_resolution` resolution algorithm can be characterized as follows (cf. Dahllöf 2002):

$$\texttt{t1\_Mary-t4\_two} < \begin{array}{l} \texttt{t5\_students} \\ \texttt{t2\_says-}\boxed{\texttt{t6\_smiled}} \end{array}$$

This reading corresponds to `tds`, `rss`, and `tree` values as follows. The `tds` value is an instance of the corresponding underspecified `tds` value:

```
[lx(t1_Mary,Mary,q1),
 body(t1_Mary,global),
 lx(t2_says,say,p_a1_para),
 v_sf(t2_says,[-,-,-]),
 c1(t1_Mary,t2_says),
 ptop(t2_says,t6_smiled),
 lx(t4_two,two,q2),
 restr(t4_two,t5_students),
 body(t4_two,t2_says),
 lx(t5_students,student,p_a1),
 c1(t4_two,t5_students),
 lx(t6_smiled,smile,p_a1),
 v_sf(t6_smiled,[+,-,-]),
 c1(t4_two,t6_smiled)]
```

The `qeqs` value is consequently simply instantiated as follows:

```
[t6_smiled qeq t6_smiled,
 t5_students qeq t5_students]
```

The `rss` TDS statements define a paratactic argument and an anchor:

```
[parg(t2_says,t6_smiled),
 anchor(t2_says,[t(t6_smiled,
                   1,
                   Ind(t6_smiled,1))])]
```

The (redundant) `tree` value gives an overview of the scopal situation and is printed as follows:

```
t4_two:q2n(
  t5_students:[],
  t2_says:ptop(
    t6_smiled:[]))
```

Scopal trees like this one are also used as a data structure by the resolution algorithm. They are Prolog terms according to this syntax:

- A term of the form *T*:`[]` represents a tree with the token *T* as the only node. That is, *T* is both a top node and a leaf node.
- A term of the form *T*:`q2n(`*T1*`,`*T2*`)` represents a tree with the two-place quantifier token *T* as the top node and the trees *T1* and *T2* representing the restriction and body subtrees, respectively.
- A term of the form *T*:`q1n(`*T1*`)` represents a tree with the one-place quantifier token *T* as the top node and the tree *T1* representing the body subtree.
- A term of the form *T*:`ptop(`*T1*`)` represents a tree with the paratactic predicate token *T* as the top node and the tree *T1* representing the paratactic argument subtree.

3.3 The Algorithm for Generating Readings

The `scopal_resolution` postparse option is used to apply an algorithm for generating the scopally possible readings from an underspecified TDS-analysis.

The `scopal_resolution` option is defined by this clause:

```
postparse_procedure(scopal_resolution,
                    FS,
                    Readings):-
    !,
    path_value(FS,
               cont:tds,
               prolog_term(Logical_form-[],_)),
    path_value(FS,
               cont:qeqs,
```

```
                prolog_term(Qeqs-[],_)),
    build_readings(Logical_form,
                   Qeqs,
                   Readings).
```

The underspecified set of TDS statements and the `qeq` statement list are retrieved. The set of possible readings is then found by the Prolog procedure `build_readings/3`. This call makes the original lists of TDS statements and `qeq` statements more specific and produces a scopal tree (`Tree`, on the form defined above).

```
build_readings(Tds_list,Qeqs,Readings):-
    findall(reading(Tds_list,Qeqs,Tree),
            build_scopal_tree(Tds_list,
                              Qeqs,
                              Tree),
            Reading_triples),
    make_FSs_for_readings(Reading_triples,
                          Readings).
```

In `build_readings/3` the procedure `build_scopal_tree/3` is used to assemble specific scopal trees from a TDS statement list and a `qeq` statement list. The readings come as represented by terms of the form `reading(`*Tds_list*`,`*Qeqs*`,`*Tree*`)`, where *Tds_list* is a TDS statement list, *Qeqs* is a `qeq` statement list, and *Tree* is the tree structure (as described above). The `reading/3`-terms are converted into feature structures by means of the procedure `make_FSs_for_readings/2`.

The main Prolog procedure behind the `scopal_resolution` option is `build_scopal_tree/3`, which is defined by this clause.

```
build_scopal_tree(Tds_list,Qeqs,Tree):-
    separate_tds_statements(Tds_list,
                            Lexical_info,
```

```
                          Scopal_info,
                          Coindexations),
    collect_scopal_nodes(Lexical_info,
                         Scopal_info,
                         Nodes),
    collect_global_quantifiers(Tds_list,
                               Global_quantifiers),
    build_scopal_tree_aux(Tree,
                          Nodes,
                          Qeqs,
                          Coindexations,
                          Global_quantifiers).
```

First, the TDS statements of `Tds_list` are separated into different categories by the procedure `separate_tds_statements/4`. Lexical statements (`lx/3`-terms), immediate outscoping statements (having `restr/2`, `body/2`, or `ptop/2` for main functor) and coindexation statements (having `c1/2` or `c2/2` for main functor) are separated into the lists `Lexical_info`, `Scopal_info`, and `Coindexations`, respectively. The predicate `collect_scopal_nodes/3` is used to collect the scopal nodes in a list (see the code in Appendix C, page 196). The scopal nodes are the nodes of the scopal tree. Quantifiers with a global scope are outside of the tree, so to speak. The scopal nodes are derived from the list of lexical statements, `Lexical_info`, and each one is of one of the forms: *T* : `q2n(`*T1*`,`*T2*`)`, *T* : `q1n(`*T1*`)`, *T* : `ptop(`*T1*`)`, or *T* : `[]`, where *T* is a token identifer and *T1* and *T2* are variables corresponding to not yet determined scopal subtrees. These nodes (collected in the list `Nodes`) are tied by means of structure sharing to the relevant immediate outscoping statements in `Scopal_info`. After that, the procedure `collect_global_quantifier_nodes/2` is used to collect the set of quantifiers with global scope. These are put in the list `Global_quantifiers`.

A valid scopal tree can then be assembled from the set of nodes

(Nodes) with the help of build_scopal_tree_aux/5. The tree has to satisfy the qeq/2-statements in Qeqs, the binding requirement, which is sensitive to the coindexation statements in Coindexations and the global quantifiers in Global_quantifiers. The result, Tree, is a fully specified scopal tree (i.e. a term of one of the forms: *T*:q2n(*T1*,*T2*), *T*:q1n(*T1*), *T*:ptop(*T1*), or *T*:[]).

The procedure build_scopal_tree_aux/5 is defined by this clause:

```
build_scopal_tree_aux(Tree,
                      Nodes,
                      Qeqs,
                      Coindexations,
                      Global_quantifiers):-
    assemble_tree(Tree,Nodes,[]),
    check_qeq_statements(Qeqs,Tree),
    check_binding_requirement(Coindexations,
                              Tree,
                              Global_quantifiers).
```

First, the procedure assemble_tree/3 assembles a tree from the given nodes. Then, it is checked that all qeq/2-statements (in Qeqs) holds and that the binding requirement is satisfied, given the coindexation statements (in Coindexations) and the set of global quantifiers (in Global_quantifiers).

The procedure assemble_tree/3 is only sensitive to the number of subtrees that are appropriate for each node. It is a straightforward recursive procedure, see Appendix C, page 204. The qeq/2-statements are checked one by one by means of the procedure check_qeq_statements/2, see Appendix C, page 205. The predicate check_binding_requirement/3 checks that in each case of coindexation between a quantifier and a predicate, the quantifier outscopes the predicate or is of global scope. Its first argument is a

list of coindexation statements, the second one the tree to be checked, and the third one the list of quantifiers with global scope (Appendix C, page 207).

The scopal resolution algorithm is intended to show that underspecification and scopal resolution work in a logically clear and simple way given the annotational TDS semantics. Computational efficiency has not been a top priority in the design of this algorithm.

4 Concluding Remarks

The present grammar and implementation of TDS covers a small fragment of English. Its analysis of syntax closely follows HPSG, but the grammar covers only a few aspects of HPSG. The purpose of the present work has been to show that the TDS semantics fits into the kind of constraint-based picture of syntax that HPSG represents. The TDS annotations derive from the lexical nodes and are collected in a simple way by means of the semantic principle associated with the phrase structure rules. The annotational character of the TDS information and the hooks-and-holes style approach to the compositional principles gives us an easy to grasp and perspicuous syntactico-semantic framework.

The scopal resolution procedure that has been described here provides a means for testing 'scopal' intuitions against the TDS semantics and the resolution principles. The grammar has not been evaluated in this way, for the simple reason that only people with a fairly sophisticated understanding of logic will be able to provide such intuitions. Nevertheless, the monotonic resolution procedure shows that a set of underspecified TDS annotations in a natural and perspicuous may be turned into a set of TDS statements defining specific truth conditions.

Several interesting questions remain to be answered, of course. First, there are several linguistic construction types that are challenging for the TDS framework, e.g. conditionals, relative clauses, and questions. Secondly, the relation between TDS annotations and the use of traditional calculi for representing truth conditions is in need of further investigation.

Another issue that is interesting in relation to a semantic framework like TDS is whether this kind of analysis may be performed without the use of an elaborate syntax. The TDS statements constitute a kind of dependency annotations. It should be possible to ascribe these more or less directly, using the kind of mechanisms associated with dependency grammar, e.g. heuristic ones. In this way it would be possible to arrive at a system for semantic 'chunking', which would able to find groups of semantically connected words, without necessarily having to parse entire sentences.

Some preliminary work has been done on generation based on the present grammar, and the results are promising. The fact that TDS annotations are decidedly surface-oriented may be a source for improved efficiency, compared to generation from other kinds of semantic representation. This aspect of the TDS framework also gives us a new way of looking at the relation between semantics and (other) language-specific properties.

References

Copestake, A., D. Flickinger, I. A. Sag, & C. Pollard, 1999, 'Minimal Recursion Semantics: An Introduction', unpublished MS, available under: `http://www.cl.cam.ac.uk/~aac10/papers/newmrs.pdf` (as of November 22, 2002).

Copestake, A., A. Lascarides, & D. Flickinger, 2001, 'An Algebra for Semantic Construction in Constraint-based Grammars'. In *Proceedings of the 39th Annual Meeting of the Association for Computational Linguistics (ACL 2001)* (Toulouse), Morgan Kaufmann Publishers, 132–139.

Dahllöf, M., 2002, 'Token Dependency Semantics and the Paratactic Analysis of Intensional Constructions', *Journal of Semantics* **19**, 333-368.

Pollard, C. & I. A. Sag, 1994, *Head-Driven Phrase Structure Grammar*, Chicago & London, The University of Chicago Press.

Sag, I.A. & T. Wasow, 1999, *Syntactic Theory: A Formal Introduction*, Stanford, CSLI Publications.

Appendix A: Sample Analyses

The sentence *Mary says that two students smiled* has two readings. This is the analysis given directly by the grammar. The semantic representation is scopally underspecified:

```
[phrase_ssh
 TOKEN t1_Mary+(t2_says+(t3_that+
           (t4_two+t5_students+t6_smiled)))
 HEAD [v
       MOD neg_s
       VFORM present
       V_SF [-,-,-]]
 SPR < >
 COMPS < >
 SLASH < >
 CONT [cont_sort
       HOOK [hole_v
             LTOP t2_says
             EXTARG t1_Mary]
       H [holes_v
          MOD_HOLE closed
          SUBJ_HOLE closed
          COMP_HOLES < >]
       TDS [lx(t1_Mary,Mary,q1),
            body(t1_Mary,global),
            lx(t2_says,say,p_a1_para),
```

```
            v_sf(t2_says,[-,-,-]),
            c1(t1_Mary,t2_says),
            ptop(t2_says,XX0),
            lx(t4_two,two,q2),
            restr(t4_two,XX1),
            body(t4_two,XX2),
            lx(t5_students,student,p_a1),
            c1(t4_two,t5_students),
            lx(t6_smiled,smile,p_a1),
            v_sf(t6_smiled,[+,-,-]),
            c1(t4_two,t6_smiled)|XX3]-XX3
       QEQS [XX0 qeq t6_smiled,
             XX1 qeq t5_students|XX4]-XX4]
 PM neg_p]
```

The two readings of the sentence *Mary says that two students smiled* are generated by the `scopal_resolution` postoperation. The name quantifier is assigned global scope.

```
[sign
 CONT [cont_sort
       TDS [lx(t1_Mary,Mary,q1),
            body(t1_Mary,global),
            lx(t2_says,say,p_a1_para),
            v_sf(t2_says,[-,-,-]),
            c1(t1_Mary,t2_says),
            ptop(t2_says,t6_smiled),
            lx(t4_two,two,q2),
            restr(t4_two,t5_students),
            body(t4_two,t2_says),
            lx(t5_students,student,p_a1),
            c1(t4_two,t5_students),
            lx(t6_smiled,smile,p_a1),
```

```
           v_sf(t6_smiled,[+,-,-]),
           c1(t4_two,t6_smiled)|XX0]-XX0
      QEQS [t6_smiled qeq t6_smiled,
            t5_students qeq t5_students|XX1]-XX1
      PARA [parg(t2_says,t6_smiled),
            anchor(t2_says,[t(t6_smiled,1,
                              Ind(t6_smiled,1))])]
      TREE t4_two:q2n(
             t5_students:[],
             t2_says:ptop(
               t6_smiled:[]))]]


[sign
 CONT [cont_sort
      TDS [lx(t1_Mary,Mary,q1),
           body(t1_Mary,global),
           lx(t2_says,say,p_a1_para),
           v_sf(t2_says,[-,-,-]),
           c1(t1_Mary,t2_says),
           ptop(t2_says,t4_two),
           lx(t4_two,two,q2),
           restr(t4_two,t5_students),
           body(t4_two,t6_smiled),
           lx(t5_students,student,p_a1),
           c1(t4_two,t5_students),
           lx(t6_smiled,smile,p_a1),
           v_sf(t6_smiled,[+,-,-]),
           c1(t4_two,t6_smiled)|XX0]-XX0
      QEQS [t4_two qeq t6_smiled,
            t5_students qeq t5_students|XX1]-XX1
      PARA [parg(t2_says,
                 t4_two+t5_students+t6_smiled),
            anchor(t2_says,[])]
```

```
       TREE t2_says:ptop(
              t4_two:q2n(
                t5_students:[],
                t6_smiled:[]))]]
```

These are a few examples of lexical entries. The verb form *says*:

```
[word_valency
 TOKEN XX0
 HEAD [v
       MOD neg_s
       VFORM present
       V_SF [-,-,-]]
 SPR <[phrase
       HEAD [nom
             AGR [agr_sort
                  NUM sing
                  PERS p3]
             CASE subc]
       SPR < >
       COMPS < >
       SLASH < >
       PM neg_p]>
 COMPS <[sign
         HEAD [verb]
         SPR < >
         COMPS < >
         PM neg_p]>
 SLASH < >
 CONT [cont_sort
       HOOK [hole_v
             LTOP XX0
             EXTARG XX1]
```

```
       H [holes_v
          MOD_HOLE closed
          SUBJ_HOLE [hole_n
                     KEY XX1]
          COMP_HOLES <[hole
                       LTOP XX2]>]
       TDS [lx(XX0,say,p_a1_para),
            v_sf(XX0,[-,-,-]),
            c1(XX1,XX0),
            ptop(XX0,XX3)|XX4]-XX4
       QEQS [XX3 qeq XX2|XX5]-XX5]
 LEXEME say]
```

The adverb *intentionally*:

```
[word_no_valency
 TOKEN XX0
 HEAD [adv
       MOD [word
            HEAD [v]
            PM neg_p]
       PREMOD yes]
 SPR < >
 COMPS < >
 SLASH < >
 CONT [cont_sort
       HOOK [hole_v
             LTOP XX0
             EXTARG XX1]
       H [holes
          MOD_HOLE [hole_v
                    LTOP XX2
                    EXTARG XX1]]
```

```
       TDS [lx(XX0,intentionally,p_a1_para),
            c1(XX1,XX0),
            ptop(XX0,XX3)|XX4]-XX4
       QEQS [XX3 qeq XX2|XX5]-XX5]
 PM neg_p
 LEXEME intentionally]
```

The adverb *probably*:

```
[word_no_valency
 TOKEN XX0
 HEAD [adv
       MOD [word
            HEAD [v]
            PM neg_p]
       PREMOD yes]
 SPR < >
 COMPS < >
 SLASH < >
 CONT [cont_sort
       HOOK [hole_v
             LTOP XX0
             EXTARG XX1]
       H [holes
          MOD_HOLE [hole_v
                    LTOP XX2
                    EXTARG XX1]]
       TDS [lx(XX0,probably,p_para),
            ptop(XX0,XX3)|XX4]-XX4
       QEQS [XX3 qeq XX2|XX5]-XX5]
 PM neg_p
 LEXEME probably]
```

Appendix B: The Grammar

This appendix gives a full listing of the grammar (`gram_tds.pl`) described in this report. Comments appear within boxed paragraphs. There is also a file with additional lexical entries (`lex_tds.pl`).

•

```
application('tds-jos',
            ['English'],
            '/home/staff/matsd/petfsg/').
```

Declaration of Types and Features

```
declaration

  [aops subsumes [          % sign
     neg_s subsumes [],     % absent sign
     sign subsumes [        % present sign
       lexitem subsumes [   % lexical item
         lxm subsumes [     % lexeme
           infl_lxm subsumes []],
                            % inflected lexeme
         word subsumes [    % surface word form
           punct_mark subsumes [
                            % punctuation marks:
             neg_p subsumes [],
```

```
                        % no punctuation mark
            full_stop subsumes [],
            comma subsumes [],
            colon subsumes [],
            semi_colon subsumes [],
            exclam subsumes [],
            questnm subsumes []],
          word_no_valency subsumes [],
                        % lacks complement
                        % valency
          word_valency subsumes []],
                        % carries complement
                        % valency
        phrase subsumes [   % phrasal signs
          phrase_wp subsumes [],
                        % phrase from word
          phrase_hc subsumes [],
                        % head-complement
          phrase_ssh subsumes [],
                        % subject-head
          phrase_dsh subsumes [],
                        % determiner
                        % specifier-head
          phrase_mh subsumes [],
                        % modifier-head
          phrase_hp subsumes [][]]],
                        % head-punctuation

  head_sort subsumes [
     verb subsumes [       % verbal
       v subsumes [],      %   ordinary pos verb
       cplzer subsumes []],
                           %   complementizer
```

```
    agrpos subsumes [
      determin subsumes [],
                          % determiner
      nom subsumes [      % nominal
        cnnoun subsumes [],
                          % common noun
        perspron subsumes [],
                          % personal pronoun
        prnoun subsumes []]],
                          % proper noun
    adj subsumes [],      % adjective
    adv subsumes [],      % adverb
    conj subsumes [],     % conjunction
    prep subsumes [       % preposition
      mk_prep subsumes [],
                          % "marker" prep.
      pred_prep subsumes []]],
                          % predicational prep.

  agr_sort subsumes [],   % agreement feature
                          % bundle

  cont_sort subsumes [],  % content structure

  holes subsumes [        % hole collection
    holes_v subsumes [],  %   for verbals
    holes_d subsumes []], %   for determiners

  hole_type subsumes [    % hole type
    closed subsumes [],   % i.e. unavailable
    hole subsumes [       % "open" hole
      hole_n subsumes [],
                          % for nominal expr.
```

```
    hole_v subsumes []]],
                          % for verbal expr.

boolean subsumes [        % true/false boolean
  no subsumes [],
  yes subsumes []],

number_sort subsumes [    % number
  sing subsumes [],
  plur subsumes []],

def_sort subsumes [       % definiteness
  def subsumes [],
  indef subsumes []],

case_sort subsumes [      % case
  nomin subsumes [        %   i.e. non-genitive
    subc subsumes [],     %   subject form
    objc subsumes []],    %   object form
  gen subsumes []],       %   genitive

pers_sort subsumes [      % person:
  p1 subsumes [],         %   first
  p2 subsumes [],         %   second
  p3 subsumes []],        %   third

vform_sort subsumes [     % verbal infl.
  finite subsumes [
    present subsumes [],
    preterit subsumes []],
  infinite subsumes [
    infinitive subsumes [],
    prp subsumes [],      % present
```

```
                          %  participle
      psp subsumes []]],  % past
                          %  participle

  v_kind_sort subsumes [  % kind of verb
    aux subsumes [],      %  auxiliary verb
    notaux subsumes []]]  %  non-auxiliary verb

where

  [sign features
     [token:prolog_term,
      head:head_sort,
      spr:list,
      comps:list,
      slash:list,
      cont:cont_sort,
      pm:punct_mark],

   lexitem features
     [lexeme:prolog_term],

   head_sort features
     [mod:aops,           % allowed to modify
      premod:boolean],    % premodifier

   agrpos features
     [agr:agr_sort],      % agreement

   nom features
     [def:def_sort,       % definiteness
      case:case_sort],    % case
```

```
verb features
  [vform:vform_sort,    % inflection
   v_kind:v_kind_sort,  % kind of verb
   v_sf:prolog_term],   % "semantic"
                        %  verbal features
prep features
  [pform:prolog_term],  % prepositions'
                        % "form"

cont_sort features
  [hook:hole_type,
   h:holes,             % holes collected
                        %  under this feature
   tds:prolog_term,     % token dependency
                        %   statements
   qeqs:prolog_term,    % qeq statements
   para:prolog_term,    % info about
                        %   paratactic args
   tree:prolog_term],   % scopal tree
                        %   structure

agr_sort features
  [num:number_sort,     % number
   pers:pers_sort],     % person

hole features
  [ltop:prolog_term],

hole_n features         % "nominal" hole
  [key:prolog_term],

hole_v features         % "verbal" hole
  [extarg:prolog_term],
```

```
   holes features
     [mod_hole:hole_type],

   holes_v features        % holes on verb
     [subj_hole:hole_type,
      comp_holes:list],

   holes_d features        % holes on determ.
     [spec_hole:hole_type]].
```

Initial Symbol Definition

```
initial_symbol
   [head:[vform:finite],  % I.e. sentence
    spr:{},
    comps:{},
    slash:{}].
```

Leaf Symbol Definition

```
leaf_symbol word.
```

The Phrase Structure Rules

Phrase from Word Rule

This rule corresponds to the case of the head-complement rule applying when the `comps` value is the empty list. It just takes us from a word to the corresponding phrase.

```
phrase_from_word rule

[phrase_wp,
```

```
 token: <>T1,
 head:xHFP,
 comps:{},
 spr:xSpr,
 slash:{},
 cont:xCont,
 pm:neg_p]                                ===>

[  [word_valency,
    token: <>T1,
    head:xHFP,
    comps:{},
    spr:xSpr,
    slash:{},
    cont:xCont,
    pm:neg_p]].
```

The Head-Complement Rule

This version of the head-complement rule combines the head with the first complement as required by the first item on the `comps` list. This means that one application of this rule per complement is required. The + operator represents physical aggregation. The `tds` and `qeqs` values are appended by means of difference list unification.

`hd_comps rule`

```
[phrase_hc,
 token: <>(T1+T2),
 head:xHFP,
 comps:xComps,
 spr:xSpr,
 cont:[hook:xHook,
       h:[subj_hole:x_Subj_Hole,
          comp_holes:x_Comp_Holes,
```

```
          mod_hole:x_Mod_Hole],
       tds: <>Tds1-Tds_open_tail,
       qeqs: <>Qeqs1-Qeqs_open_tail],
 pm:neg_p]                              ===>

[  [token: <>T1,
    head:xHFP,
    spr:xSpr,
    comps:xComp^xComps,
    cont:[hook:xHook,
          h:[subj_hole:x_Subj_Hole,
             comp_holes:x_Comp_Hole^x_Comp_Holes,
             mod_hole:x_Mod_Hole],
          tds: <>Tds1-Tds2,
          qeqs: <>Qeqs1-Qeqs2],
    pm:neg_p],

   [xComp,
    token: <>T2,
    cont:[hook:x_Comp_Hole,
          tds: <>Tds2-Tds_open_tail,
          qeqs: <>Qeqs2-Qeqs_open_tail],
    pm:neg_p]].
```

The Specifier-Head Rule, Subject Version

The grammatical head (the verb phrase) is the semantic head here. Its `subj_hole` matches the subject (the specifier).

```
subj_spr_hd rule

[phrase_ssh,
 token: <>(T1+T2),
 head:xHFP,
```

```
 spr:{},
 comps:{},
 cont:[hook:xHook,
       h:[subj_hole:closed,
          comp_holes:x_Comp_Holes,
          mod_hole:x_Mod_Hole],
       tds: <>Tds1-Tds_open_tail,
       qeqs: <>Qeqs1-Qeqs_open_tail],
 pm:neg_p]                              ===>

[  [xSpr,
    token: <>T1,
    cont:[hook:x_Subj_Hole,
          tds: <>Tds1-Tds2,
          qeqs: <>Qeqs1-Qeqs2],
    pm:neg_p],

   [phrase,
    token: <>T2,
    head:xHFP,
    spr:{xSpr},
    comps:{},
    cont:[hook:xHook,
          h:[subj_hole:x_Subj_Hole,
             comp_holes:x_Comp_Holes,
             mod_hole:x_Mod_Hole],
          tds: <>Tds2-Tds_open_tail,
          qeqs: <>Qeqs2-Qeqs_open_tail],
    pm:neg_p]].
```

The Specifier-Head Rule, Determiner Version

The determiner is taken to be the semantic head here, and the (nominal) syntactic head matches the determiner's `spec_hole`. Note that determiner-noun agreement is implemented by means of the `agr` feature, and is taken to be a lexical fact about nouns, in the style of Sag & Wasow (1999: 92). Pollard's and Sag's (1994: 51) 'Spec Principle' is not put to use in this grammar.

```
det_spr_hd rule

[phrase_dsh,
 token: <>(T1+T2),
 head:xHFP,
 spr:{},
 comps:{},
 cont:[hook:xHook,
       h:[spec_hole:closed,
          mod_hole:x_Mod_Hole],
       tds: <>Tds1-Tds_open_tail,
       qeqs: <>Qeqs1-Qeqs_open_tail],
 pm:neg_p]                              ===>

 [[xSpr,
     token: <>T1,
     head:determin,
     cont:[hook:xHook,
           h:[spec_hole:x_Spec_Hook,
              mod_hole:x_Mod_Hole],
           tds: <>Tds1-Tds2,
           qeqs: <>Qeqs1-Qeqs2],
     pm:neg_p],

   [phrase,
```

```
    token: <>T2,
    head:xHFP,
    spr:{xSpr},
    comps:{},
    cont:[hook:x_Spec_Hook,
          tds: <>Tds2-Tds_open_tail,
          qeqs: <>Qeqs2-Qeqs_open_tail],
    pm:neg_p]].
```

The Premodifier-Head Rule

The `head:premod` value (boolean) on the modifier defines it as a premodifier.

```
mod_hd rule

[phrase_mh,
 token: <>(T1+T2),
 head:xHFP,
 spr:xSpr,
 comps:xComps,
 cont:[hook:x_Mod_Hook,
       h:[subj_hole:x_Subj_Hole,
          comp_holes:x_Comp_Holes,
          mod_hole:x_Mod_Hole],
       tds: <>Tds1-Tds_open_tail,
       qeqs: <>Qeqs1-Qeqs_open_tail],
 pm:neg_p]                          ===>


[  [token: <>T1,
    head:[mod:xHead,
          premod:yes],
    spr:{},
```

```
    comps:{},
    cont:[hook:x_Mod_Hook,
          h:[mod_hole:x_Head_Hook],
          tds: <>Tds1-Tds2,
          qeqs: <>Qeqs1-Qeqs2],
    pm:neg_p],

   [xHead,
    token: <>T2,
    head:xHFP,
    spr:xSpr,
    comps:xComps,
    cont:[hook:x_Head_Hook,
          h:[subj_hole:x_Subj_Hole,
             comp_holes:x_Comp_Holes,
             mod_hole:x_Mod_Hole],
          tds: <>Tds2-Tds_open_tail,
          qeqs: <>Qeqs2-Qeqs_open_tail],
    pm:neg_p]].
```

Phrase-Punctuation-Mark Rule

This rule incorporates a punctuation mark in a phrase: A phrase with `pm:neg_p` combines with a punctuation mark and forms a phrase with the feature `pm` specifying the sister punctuation mark. The mother and daughter phrases otherwise carry the same feature values.

```
phr_punct rule

[phrase_hp,
 token: <>Token,
 head:xHd,
 spr:{},
 comps:{},
 cont:xCont,
```

```
 pm:xPM]                                    ===>

[  [phrase,
    token: <>Token,
    head:xHd,
    spr:{},
    comps:{},
    cont:xCont,
    pm:neg_p],

   [xPM,
    punct_mark]].
```

Lexical Information

General Definitions

```
content_lexitem is_short_for
  [token: <>Token,
   cont:[hook:[ltop: <>Token]],
   slash:{},
   pm:neg_p].
```

`content_lexitem` applies to lexical entries which contribute to semantics. The feature values `token` and `cont:hook:ltop` are identified.

```
np_subj is_short_for
   [phrase,
    head:[nom,
          case:subc],
    spr:{},
    comps:{},
    slash:{},
    pm:neg_p].
```

Subject form NPs.

```
np_obj is_short_for
   [phrase,
    head:[nom,
          case:objc],
    spr:{},
    comps:{},
    slash:{},
    pm:neg_p].
```

Object form NPs.

```
pp_obj is_short_for
   [phrase,
    head:mk_prep,
    spr:{},
    comps:{},
    slash:{},
    pm:neg_p].
```

Prepositional object PPs.

```
inf_phrase_bare is_short_for
   [head:[v,
          vform:infinitive],
    comps:{},
    pm:neg_p].
```

Infinitival phrases without a *to* complementizer.

```
inf_phrase_to is_short_for
   [head:[cplzer,
          vform:infinitive],
    comps:{},
    pm:neg_p].
```

Infinitival phrases with a *to* complementizer.

```
prp_phrase is_short_for
   [head:[v,
          vform:prp],
    comps:{},
    pm:neg_p].
```

Present participle phrases.

```
psp_phrase is_short_for
   [head:[v,
          vform:psp],
    comps:{},
    pm:neg_p].
```

Past participle phrases.

```
no_mod is_short_for
   [head:[mod:neg_s],
    cont:[h:[mod_hole:closed]]].
```

For signs which can not occur as (pre- or post-) modifiers.

Proper Nouns

```
proper_noun is_short_for
   [word_valency,
    nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[prnoun,
          agr:[num:sing,
               pers:p3]],
```

```
    spr:{},
    comps:{},
    nfs no_mod,
    cont:[hook:[key: <>Token],
          tds: <>[lx(Token,Lexeme,q1),
                  body(Token,global)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].

'Galileo' >>>
   [nfs proper_noun,
    lexeme: <>'Galileo'].
```

Personal Pronouns

```
personal_pronoun is_short_for
   [nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:perspron,
    spr:{},
    comps:{},
    nfs no_mod,
    cont:[hook:[key: <>Token],
          tds: <>[lx(Token,Lexeme,q1),
                  body(Token,global)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].

he >>>
   [word_valency,
    nfs personal_pronoun,
    head:[agr:[num:sing,
               pers:p3]],
    lexeme: <>'he'].
```

Determiners

```
determiner is_short_for
   [nfs content_lexitem,
    word_no_valency,
    token: <>Token,
    lexeme: <>Lexeme,
    head:determin,
    spr:{},
    comps:{},
    nfs no_mod,
    cont:[hook:[key: <>Token],
          h:[spec_hole:[key: <>Token,
                        ltop: <>LTop]],
          tds: <>[lx(Token,Lexeme,q2),
                  restr(Token,Restr),
                  body(Token,_)|Tds]-Tds,
          qeqs: <>[Restr qeq LTop|Qeqs]-Qeqs]].

many >>>
   [nfs determiner,
    head:[agr:[num:plur]],
    lexeme: <>'many'].

the >>>
   [nfs determiner,
    head:[agr:[num:sing]],
    lexeme: <>'the_sing'].

the >>>
   [nfs determiner,
    head:[agr:[num:plur]],
    lexeme: <>'the_plur'].
```

By having two senses for the determiner *the*, we allow number to be reflected (lexically) in the `lexeme` value.

Noun Morphology

```
infl_reg(sing_noun_infl,id).
infl_reg(plur_noun_infl,affx(s)).
```

The morphological operation producing plural forms is defined as affixation of an *s* (‘`affx(s)`’).

```
lexrule sing_noun
  morph  sing_noun_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                cnnoun],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          token:x0,
          lexeme:x1,
          head:[x2,
                cnnoun,
                agr:[num:sing]],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule plur_noun
```

```
morph  plur_noun_infl
input  [infl_lxm,
        token:x0,
        lexeme:x1,
        head:[x2,
              cnnoun],
        spr:x3,
        comps:x4,
        slash:x5,
        cont:x6]
output [word_valency,
        token:x0,
        lexeme:x1,
        head:[x2,
              cnnoun,
              agr:[num:plur]],
        spr:x3,
        comps:x4,
        slash:x5,
        cont:x6].
```

The common kind of singular/plural noun inflection in English.

Common Nouns Requiring a Specifier

Sag & Wasow's (1999: 92) Nominal SPR Agreement (NSA) constraint is hard-coded into this NFS. The variable `xAgr` is used to express the sharing of `agr` values between the determiner and the noun.

```
common_noun_spr is_short_for
   [nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[cnnoun,
```

```
          agr:[xAgr,       %%% NSA
               pers:p3]],
    spr:{[head:[determin,
                agr:xAgr], %%% NSA
          pm:neg_p]},
    comps:{},
    nfs no_mod,
    cont:[hook:[key: <>SprKey],
          h:[subj_hole:closed],
          tds: <>[lx(Token,Lexeme,p_a1),
                  c1(SprKey,Token)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].
```

This does not give us bare plurals.

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

child >>>
   [infl_lxm,
    nfs common_noun_spr,
    lexeme: <>child].

suppletive_form(child,plur_noun_infl,children).
```

Complementizers

```
cplzer is_short_for
   [word_valency,
    slash:{},
    head:[cplzer,
          vform:xVform,
```

```
          v_sf:xVSF],
    nfs no_mod,
    comps:{[head:[v,
                  vform:xVform,
                  v_sf:xVSF],
            comps:{}]},
    cont:[hook:x_Comp_Hole,
          h:[comp_holes:{x_Comp_Hole}],
          tds: <>Tds-Tds,
          qeqs: <>Qeqs-Qeqs]].
```

A complementizer takes a VP complement. The resulting `cplzer` phrase shares all its properties with the the complement, except its `head` category.

```
that >>>
   [nfs cplzer,
    spr:{},
    comps:{[head:[vform:finite],
            spr:{}]}].

to >>>
   [nfs cplzer,
    spr:{xSpr},
    comps:{[head:[vform:infinitive],
            spr:{xSpr}]}].
```

Prepositions

```
mk_preposition is_short_for
   [word_valency,
    slash:{},
    head:mk_prep,
    nfs no_mod,
```

```
    comps:{nfs np_obj},
    spr:{},
    cont:[hook:x_Comp_Hole,
          h:[comp_holes:{x_Comp_Hole}],
          tds: <>Tds-Tds,
          qeqs: <>Qeqs-Qeqs]].

at >>>
   [nfs mk_preposition,
    head:[pform: <>at]].
```

Verbs: General Aspects

```
verb_general is_short_for
   [nfs content_lexitem,
    head:v,
    nfs no_mod].
```

Verb Form Abbreviations

```
present_third_singular is_short_for
   [head:[vform:present,
          v_sf: <>['-','-','-']],
    spr:{[head:[agr:[num:sing,
                     pers:p3]]]}].

present_third_plural is_short_for
   [head:[vform:present,
          v_sf: <>['-','-','-']],
    spr:{[head:[agr:[num:plur,
                     pers:p3]]]}].

preterit_third_singular is_short_for
  [head:[vform:present,
```

```
         v_sf: <>['+','-','-']],
   spr:{[head:[agr:[num:sing,
                   pers:p3]]]}].

preteritum is_short_for
   [head:[vform:preterit,
          v_sf: <>['+','-','-']]].

infinitive_form is_short_for
   [head:[vform:infinitive,
          v_sf: <>[n,'-','-']]].

prp_form is_short_for
   [head:[vform:prp,
          v_sf: <>[_,_,'+']]].

psp_form is_short_for
   [head:[vform:psp,
          v_sf: <>[_,'+','-']]].
```

Verb Inflection

```
lexrule v_infl_infin
  morph  v_infin_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
```

```
          nfs infinitive_form,
          token:x0,
          lexeme:x1,
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule v_infl_third_plural
  morph v_third_plural_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          nfs present_third_plural,
          token:x0,
          lexeme:x1,
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule v_infl_third_singular
  morph  v_third_singular_infl
  input  [infl_lxm,
```

```
          token:x0,
          lexeme:x1,
          head:[x2,
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          nfs present_third_singular,
          token:x0,
          lexeme:x1,
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule v_infl_pret
  morph  v_pret_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          nfs preteritum,
          token:x0,
          lexeme:x1,
```

```
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule v_infl_psp
  morph  v_psp_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          nfs psp_form,
          token:x0,
          lexeme:x1,
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

lexrule v_infl_prp
  morph  v_prp_infl
  input  [infl_lxm,
          token:x0,
          lexeme:x1,
          head:[x2,
```

```
                verb],
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6]
  output [word_valency,
          nfs prp_form,
          token:x0,
          lexeme:x1,
          head:x2,
          spr:x3,
          comps:x4,
          slash:x5,
          cont:x6].

infl_reg(v_infin_infl,id).
infl_reg(v_third_plural_infl,id).
infl_reg(v_third_singular_infl,affx(s)).
infl_reg(v_pret_infl,affx_ch_e(ed)).
infl_reg(v_psp_infl,affx_ch_e(ed)).
infl_reg(v_prp_infl,affx_ch_e(ing)).
```

Intransitive Verbs

```
verb_intransitive is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey]],
          tds: <>[lx(Token,Lexeme,p_a1),
```

```
                v_sf(Token,VSF),
                c1(SubjKey,Token)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].

smile >>>
   [infl_lxm,
    nfs verb_intransitive,
    lexeme: <>smile].

sleep >>>
   [infl_lxm,
    nfs verb_intransitive,
    lexeme: <>sleep].

suppletive_form(sleep,v_pret_infl,slept).
suppletive_form(sleep,v_psp_infl,slept).
```

Ordinary Transitive Verbs

```
verb_transitive is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{[nfs np_obj]},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[key: <>ObjKey]}],
          tds: <>[lx(Token,Lexeme,p_a1_a2),
                  v_sf(Token,VSF),
                  c1(SubjKey,Token),
                  c2(ObjKey,Token)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].
```

```
help >>>
   [infl_lxm,
    nfs verb_transitive,
    lexeme: <>help].
```

Prepositional Object Verbs

```
verb_prep_transitive is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{[nfs pp_obj]},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[key: <>ObjKey]}],
          tds: <>[lx(Token,Lexeme,p_a1_a2),
                  v_sf(Token,VSF),
                  c1(SubjKey,Token),
                  c2(ObjKey,Token)|Tds]-Tds,
          qeqs: <>Qeqs-Qeqs]].

smile >>>
   [infl_lxm,
    nfs verb_prep_transitive,
    comps:{[head:[pform: <>at]]},
    lexeme: <>smile_at].
```

Paratactic Verbs

```
verb_paratactic is_short_for
   [nfs verb_general,
```

```
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{[head:verb,
            comps:{},
            spr:{},
           pm:neg_p]},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[ltop: <>LTop]}],
          tds: <>[lx(Token,Lexeme,p_a1_para),
                  v_sf(Token,VSF),
                  c1(SubjKey,Token),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].

claim >>>
    [infl_lxm,
     nfs verb_paratactic,
     lexeme: <>claim].
```

Subject Raising Verbs

```
verb_subj_raising is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[xSpr,
          nfs np_subj]},
    comps:{[spr:{xSpr},
            pm:neg_p]},
    cont:[hook:[extarg: <>SubjKey],
```

```
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[ltop: <>LTop,
                          extarg: <>SubjKey]}],
          tds: <>[lx(Token,Lexeme,p_para),
                  v_sf(Token,VSF),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

Subject Raising Verbs (Bare)

Verbs like these require for complement an infinitival phrase without a *to* complementizer.

```
verb_subj_raising_bare is_short_for
   [nfs verb_subj_raising,
    comps:{nfs inf_phrase_bare}].

would >>>
   [word_valency,
    nfs verb_subj_raising_bare,
    nfs preteritum,
    lexeme: <>would].
```

Subject Raising Verbs (Non-Bare)

These verbs combine with an infinitival phrase with a *to* complementizer.

```
verb_subj_raising_to is_short_for
   [nfs verb_subj_raising,
    comps:{nfs inf_phrase_to}].

ought >>>
   [word_valency,
    nfs verb_subj_raising_to,
    nfs preteritum,
    lexeme: <>ought].
```

Subject Control Verbs

```
verb_subj_control is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[xSpr,
         nfs np_subj]},
    comps:{[nfs inf_phrase_to,
            spr:{xSpr},
            pm:neg_p]},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[ltop: <>LTop,
                          extarg: <>SubjKey]}],
          tds: <>[lx(Token,Lexeme,p_a1_para),
                  v_sf(Token,VSF),
                  c1(SubjKey,Token),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

The verb EP and the complement EP are coindexed with respect to their first argument positions. The complement is an infinitival phrase with a *to* complementizer.

```
intend >>>
    [infl_lxm,
     nfs verb_subj_control,
     lexeme: <>intend].
```

Object Raising Verbs

```
verb_obj_raising is_short_for
```

```
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{[nfs np_obj],
           [nfs inf_phrase_to,
            spr:{[]}]},
    cont:[hook:[extarg: <>SubjKey],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[key: <>ObjKey],
                         [ltop: <>LTop,
                          extarg: <>ObjKey]}],
          tds: <>[lx(Token,Lexeme,p_a1_para),
                  v_sf(Token,VSF),
                  c1(SubjKey,Token),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

The object controls the subject argument position on the infinitival complement through the `extarg` feature, but does not bind any argument position on the main verb.

```
expect >>>
    [infl_lxm,
     nfs verb_obj_raising,
     lexeme: <>expect].
```

Object Control Verbs

```
verb_obj_control is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
```

```
    head:[v_sf: <>VSF],
    spr:{[nfs np_subj]},
    comps:{[nfs np_obj],
           [nfs inf_phrase_to,
            spr:{[]}]},
    cont:[hook:[extarg: <>SubjKey],
         h:[subj_hole:[key: <>SubjKey],
            comp_holes:{[key: <>ObjKey],
                        [ltop: <>LTop,
                         extarg: <>ObjKey]}],
           tds: <>[lx(Token,Lexeme,p_a1_a2_para),
                   v_sf(Token,VSF),
                   c1(SubjKey,Token),
                   c2(ObjKey,Token),
                   ptop(Token,PTop)|Tds]-Tds,
           qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

The object controls the subject argument position on the infinitival complement through the `extarg` feature, while also binding the second argument position on the main verb.

```
persuade >>>
    [infl_lxm,
     nfs verb_obj_control,
     lexeme: <>persuade].
```

Auxiliary Verbs (Tentative Treatments)

```
verb_aux is_short_for
   [word_valency,
    slash:{},
    nfs no_mod,
    head:[v,
          v_kind:aux],
```

```
    comps:{[]},
    cont:[hook:[extarg: <>SubjKey,
                ltop: <>LTop],
          h:[subj_hole:[key: <>SubjKey],
             comp_holes:{[ltop: <>LTop,
                          extarg: <>SubjKey]}],
          tds: <>Tds-Tds,
          qeqs: <>Qeqs-Qeqs]].
```

These verbs contribute semantically by specifying the `v_sf` features on the main verb.

Temporal have

```
verb_temp_have is_short_for
   [nfs verb_aux,
    spr:{[xSubj]},
    comps:{[nfs psp_phrase,
            spr:{xSubj}]}].

have >>>
     [nfs verb_temp_have,
      nfs infinitive_form,
      comps:{[head:[v_sf: <>['n',_,_]]]}].

has >>>
     [nfs verb_temp_have,
      nfs present_third_singular,
      comps:{[head:[v_sf: <>['-',_,_]]]}].

have >>>
     [nfs verb_temp_have,
      nfs present_third_plural,
      comps:{[head:[v_sf: <>['-',_,_]]]}].
```

```
had >>>
     [nfs verb_temp_have,
      nfs preteritum,
      comps:{[head:[v_sf: <>['+',_,_]]]}].
```

Progressive be

```
verb_progr_be is_short_for
   [word_valency,
    nfs verb_aux,
    spr:{[xSubj]},
    comps:{[nfs prp_phrase,
            spr:{xSubj}]}].

is >>>
     [nfs verb_progr_be,
      nfs present_third_singular,
      comps:{[head:[v_sf: <>['-','-',_]]]}].

was >>>
     [nfs verb_progr_be,
      nfs preterit_third_singular,
      comps:{[head:[v_sf: <>['+','-',_]]]}].

be >>>
     [nfs verb_progr_be,
      nfs infinitive_form,
      comps:{[head:[v_sf: <>['n','-',_]]]}].

been >>>
     [nfs verb_progr_be,
      nfs psp_form,
      head:[v_sf: <>[Tense,_,_]],
```

```
      comps:{[head:[v_sf: <>[Tense,+,_]]]}].
```

Adjectives (Tentative Treatment)

```
adj_paratactic is_short_for
   [word_valency,
    nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:adj,
    spr:{},
    comps:{},
    head:[mod:[head:cnnoun,
               comps:{},
               pm:neg_p],
          premod:yes],
    cont:[hook:[key: <>ModKey],
          h:[mod_hole:[key: <>ModKey,
                       ltop: <>LTop]],
          tds: <>[lx(Token,Lexeme,p_a1_para),
                  c1(ModKey,Token),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

This gives us a paratactic analysis of adjectives in attributive (modifier) position. They are analyzed as two-place relations holding of an entity and a description provided by the modified noun predication token.

```
false >>>
     [nfs adj_paratactic,
      lexeme: <>false].
```

Sentence Adverbs

```
adv_sent_fs is_short_for
   [word_no_valency,
    nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:adv,
    spr:{},
    comps:{},
    head:[mod:[word,
               head:v,
               pm:neg_p],
          premod:yes],
    cont:[hook:[extarg: <>SprKey],
          h:[mod_hole:[ltop: <>LTop,
                       extarg: <>SprKey]],
          tds: <>[lx(Token,Lexeme,p_para),
                  ptop(Token,PTop)|Tds]-Tds,
          qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].

consequently >>>
     [nfs adv_sent_fs,
      lexeme: <>consequently].
```

Subject-Relative Adverbs

```
adv_sent_subj is_short_for
   [word_no_valency,
    nfs content_lexitem,
    token: <>Token,
    lexeme: <>Lexeme,
    head:adv,
    spr:{},
    comps:{},
    head:[mod:[word,
```

```
              head:v,
              pm:neg_p],
        premod:yes],
    cont:[hook:[extarg: <>SprKey],
        h:[mod_hole:[ltop: <>LTop,
                     extarg: <>SprKey]],
        tds: <>[lx(Token,Lexeme,p_a1_para),
                c1(SprKey,Token),
                ptop(Token,PTop)|Tds]-Tds,
        qeqs: <>[PTop qeq LTop|Qeqs]-Qeqs]].
```

Only word-level verbs are allowed to be modified.

```
intentionally >>>
     [nfs adv_sent_subj,
      lexeme: <>intentionally].
```

Interface Information

These pieces of information do not belong to the grammar, properly speaking, but define the resources necessary for using the grammar in a PETFSG application. Some of these clauses also define the appearance of various grammar output.

```
postparse_procedures([cont,scopal_resolution]).
```

Procedure `cont`: Only `cont`(ent) information is selected for output.
`scopal_resolution`: To perform scopal resolution.

Colour Specifications

```
type_color(sign,'#FFFFE0').
type_color(word,'#FAEBD7').
type_color(lxm,'#FA8072').
type_color(infl_lxm,'#D8BFD8').
```

```
type_color(phrase,'#FFE4B5').
type_color(head_sort,' #FFFFCC').
type_color(determin,'#FFE4B5').
type_color(nom,'#8FBC8F').
type_color(cnnoun,'#8FBC8F').
type_color(prnoun,'#8FBC8F').
type_color(perspron,'#8FBC8F').
type_color(adj,'#CCFF00').
type_color(adv,'#CCFF00').
type_color(verb,'#FFFFCC').
type_color(v,'#FFFFCC').
type_color(cplzer,'#FFC0CB').
type_color(mod_sort,'#E6E6FA').
type_color(cont_sort,'#F5FFFA').
```

Display colours associated with the various types.

Start Page Information

```
start_page_section([
   '<p>An implementation of Token Dependency ',
   'Semantics for a Fragment of English. ',
   'See documentation ',
   '<a href="http://stp.ling.uu.se/',
   '~matsd/tds/jos/">here</a>.</p>']).
```

CSS

```
css_url(
'http://stp.ling.uu.se/~matsd/petfsg/II.2/css/pgt.css').
```

•

Appendix C: Auxiliary Prolog Procedures

This is a listing of the additional Prolog procedures of which the application makes use. The file name is `application_procedures.pl` and its content is loaded into the `this_application` module, as requested in the PETFSG-II source file `this_application.pl`.

•

```
:- use_module(library(fs_handling),
              [path_value/3]).

:- use_module(library(lists)).
```

The symbol for *equality modulo quantifiers,* Copestake *et al* (2001) style:

```
:- op(300,xfy,qeq).
```

`label_token/3`

The procedure `label_token/3` is called just before a lexical inactive edge is stored in the chart. It assigns a unique identifier to each token, i.e. to the `token` feature of the feature structure associated with an elementary token of the input string. For instance, a token of *house* at position 3 is assigned the identifier `t3_house`.

```
label_token(Description,Number,Lexical_atom):-
    name(Number,Number_name),
    name(Lexical_atom,Lexical_atom_name),
    append([116|Number_name],      %% ascii for t
           [95|Lexical_atom_name], %% ascii for _
           New_name),
    name(Token_identifier,New_name),
    path_value(Description,token,
               prolog_term(Token_identifier,_)).
```

call_prolog/1

The procedure `call_prolog/1` provides the interface between the phrase structure rules and general Prolog computation. It is not used by the present grammar. This is just an error handling case.

```
call_prolog(X):-
    !,
    make_html(tx('Unexpected prolog call'(X))),
    fail.
```

atom_reshaping/3

The procedure `atom_reshaping/3` relates an atom reshaping function identifier, an input textual form, and an output textual form. See the PETFSG documentation.

```
atom_reshaping(id,Atom,Atom).
```

The identity reshaping operation.

```
atom_reshaping(affx(Affix),
               Shape_in,
               Shape_out):-
    name(Shape_in,
         Shape_in_name),
```

```
    name(Affix,
         Affix_name),
    append(Shape_in_name,
           Affix_name,
           Shape_out_name),
    name(Shape_out,
         Shape_out_name).
```

Plain affixation.

```
atom_reshaping(affx_ch_e(Affix),
               Shape_in,
               Shape_out):-
    name(Shape_in,
         Shape_in_name),
    remove_final_e(Shape_in_name,
                   Shape_in_name_e_removed),
    name(Affix,
         Affix_name),
    append(Shape_in_name_e_removed,
           Affix_name,
           Shape_out_name),
    name(Shape_out,
         Shape_out_name).
```

Affixation with removal of possible final *e* on the stem.

`remove_final_e/2`

The procedure `remove_final_e/2` removes a final 101 (ascii for e) in a list, if there is one.

```
remove_final_e(Shape_name,
               Shape_name_e_removed):-
    reverse(Shape_name,
            [101|Name_tail]),
    !,
    reverse(Name_tail,
            Shape_name_e_removed).

remove_final_e(Shape_name,
               Shape_name).
```

postparse_procedure/3

The predicate `postparse_procedure/3` defines Prolog procedures which are applied to the output FSs after the parsing process. The first argument is the name (a Prolog atom) identifying a procedure. The second argument corresponds to the input FS and the third argument to the list (set) of FSs that will be associated with the input FS by the procedure. (Also see the PETFSG documentation.)

```
postparse_procedure(cont,
                    Input_fs,
                    [Selected_fs]):-
    !,
    path_value(Input_fs,cont,Content_value),
    path_value(Selected_fs,cont,Content_value).
```

Only `cont` selected for output.

```
postparse_procedure(scopal_resolution,
                    FS,
                    Readings):-
    !,
    path_value(FS,
```

```
                cont:tds,
                prolog_term(Logical_form-[],_)),
    path_value(FS,
                cont:qeqs,
                prolog_term(Qeqs-[],_)),
    build_readings(Logical_form,
                    Qeqs,
                    Readings).
```

Finds the set of scopally resolved readings.

```
postparse_procedure(Operation,_,[]):-
  !,
  make_html(txbi(['No postparse op. applied.',
                  'Option "', Operation,
                  '" undefined. ',])).
```

This is an error case.

build_readings/3

In the procedure build_readings/3 the procedure build_scopal_tree/3 is used to assemble specific scopal trees from a TDS statement list and a qeq statement list. The readings come as represented by reading/3-terms and are converted into feature structures by means of the procedure make_FSs_for_readings/2.

```
build_readings(Tds_list,Qeqs,Readings):-
    findall(reading(Tds_list,Qeqs,Tree),
            build_scopal_tree(Tds_list,
                              Qeqs,
                              Tree),
            Reading_triples),
    make_FSs_for_readings(Reading_triples,
                          Readings).
```

`build_scopal_tree/3`

The main procedure for building scopal trees is `build_scopal_tree/3`. First, the TDS statements of `Tds_list` are separated into different categories by the procedure `separate_tds_statements/3`. Lexical statements (`lx/3`-terms), immediate outscoping statements (having `restr/2`, `body/2`, or `ptop/2` for main functor), and coindexation statements (having `c1/2` or `c2/2` for main functor) are separated into the lists `Lexical_info`, `Scopal_info`, and `Coindexations`, respectively. The predicate `collect_scopal_nodes/3` is used to collect the scopal nodes in a list. Scopal nodes are derived from the list of lexical statements, `Lexical_info`, and each one is of one of the forms: *T*:`q2n(`*T1*`,`*T2*`)`, *T*:`q1n(`*T1*`)`, *T*:`ptop(`*T1*`)`, or *T*:`[]`, where *T* is a token identifer and *T1* and *T2* are variables corresponding to not yet determined scopal subtrees. These nodes (collected in the list `Nodes`) are tied by means of structure sharing to the relevant immediate outscoping statements in `Scopal_info`. After that, the procedure `collect_global_quantifiers/2` is used to collect the set of quantifiers with global scope. These are put in the list `Global_quantifiers`. A valid scopal tree can then be assembled from the set of nodes (`Nodes`) with the help of `build_scopal_tree_aux/5`. The tree has to satisfy the `qeq/2`-statements in `Qeqs`, the binding requirement, which is sensitive to the coindexation statements in `Coindexations` and the global quantifiers in `Global_quantifiers`. The result, `Tree`, is a fully specified scopal tree.

```
build_scopal_tree(Tds_list,Qeqs,Tree):-
    separate_tds_statements(Tds_list,
                            Lexical_info,
                            Scopal_info,
                            Coindexations),
    collect_scopal_nodes(Lexical_info,
                         Scopal_info,
```

```
                          Nodes),
    collect_global_quantifiers(Tds_list,
                               Global_quantifiers),
    build_scopal_tree_aux(Tree,
                          Nodes,
                          Qeqs,
                          Coindexations,
                          Global_quantifiers).
```

build_scopal_tree_aux/5

The predicate `build_scopal_tree_aux/5` is a help procedure to the previous one and it builds a scopal tree from given sets of nodes, `qeq` statements, coindexation statements, and global quantifiers. The procedure `assemble_tree/3` assembles a tree from the given nodes. This procedure is only sensitive to the number of subtrees that is appropriate for each node. Then, it is checked that all `qeq/2`-statements (in `Qeqs`) holds and that the binding requirement is satisfied, given the coindexation statements (in `Coindexations`) and the set of global quantifiers (in `Global_quantifiers`).

```
build_scopal_tree_aux(Tree,
                      Nodes,
                      Qeqs,
                      Coindexations,
                      Global_quantifiers):-
    assemble_tree(Tree,Nodes,[]),
    check_qeq_statements(Qeqs,Tree),
    check_binding_requirement(Coindexations,
                              Tree,
                              Global_quantifiers).
```

`collect_scopal_nodes/3`
The procedure `collect_scopal_nodes/3` collects the list of scopal nodes (third argument) from the lists containing lexical information (first argument) and scopal TDS statements (second argument).

```
collect_scopal_nodes([lx(Node,_,q1)|Tail],
                     Scopal_info,
                     [Node:q1n(Body)|
                        Scopal_nodes]):-
    make_q1_node(Node,
                 Scopal_info,
                 Node:q1n(Body)),
    \+(Body==global),
    !,
    collect_scopal_nodes(Tail,
                         Scopal_info,
                         Scopal_nodes).
```

This case will never apply if one-place quantifiers have a lexicalized global body scope.

```
collect_scopal_nodes([lx(Node,_,q2)|Tail],
                     Scopal_info,
                     [Node:q2n(Restr,Body)|
                           Scopal_nodes]):-
    make_q2_node(Node,
                 Scopal_info,
                 Node:q2n(Restr,Body)),
    \+(Body==global),
    !,
    collect_scopal_nodes(Tail,
                         Scopal_info,
                         Scopal_nodes).
```

This should alway be true for the present small fragment of English, as there are no two-place quantifiers with a lexicalized global body scope in it.

```
collect_scopal_nodes(
           [lx(Node,_,Logical_valency)|Tail],
           Scopal_info,
           [Node:ptop(Paratactic_argument)|
              Scopal_nodes]):-
    paratactic_valency(Logical_valency),
    make_paratactic_node(
           Node,
           Scopal_info,
           Node:ptop(Paratactic_argument)),
    \+(Paratactic_argument==global),  %% (true)
    !,
    collect_scopal_nodes(Tail,
                         Scopal_info,
                         Scopal_nodes).
```

Paratactic predicate tokens dominate a paratactic argument subtree.

```
collect_scopal_nodes([lx(Node,
                         _,
                         Logical_valency)|Tail],
                     Scopal_info,
                     [Node:[]|Scopal_nodes]):-
    predicate_valency(Logical_valency),
    \+paratactic_valency(Logical_valency),
    !,
    collect_scopal_nodes(Tail,
                         Scopal_info,
                         Scopal_nodes).
```

Non-paratactic predicate tokens are scopal leaves.

```
collect_scopal_nodes([_|Tail],
                     Scopal_info,
                     Scopal_nodes):-
    !,
    collect_scopal_nodes(Tail,
                         Scopal_info,
                         Scopal_nodes).

collect_scopal_nodes([],_,[]).
```

collect_global_quantifiers/2

The procedure `collect_global_quantifiers/2` is used to collect the set of quantifiers with global scope.

```
collect_global_quantifiers([],[]).

collect_global_quantifiers(
            [body(Quantifier,Scope)|Tds_list],
            [Quantifier|Quantifiers]):-
    Scope==global,
    !,
    collect_global_quantifiers(Tds_list,
                               Quantifiers).

collect_global_quantifiers([_|Tds_list],
                           Quantifiers):-
    !,
    collect_global_quantifiers(Tds_list,
                               Quantifiers).
```

predicate_valency/1

The predicate `predicate_valency/1` is true of the logical valency identifiers which stand for predicates.

```
predicate_valency(P):-
    member(P,[p_a1,p_a1_a2,p_para,
              p_a1_para,p_a1_a2_para]).
```

paratactic_valency/1

The predicate `paratactic_valency/1` is true of the logical valency identifiers which stand for paratactic predicates.

```
paratactic_valency(P):-
    member(P,[p_para,p_a1_para,p_a1_a2_para]).
```

make_q1_node/3

The procedure `make_q1_node/3` collects the information pertaining to a one-place quantifier subtree (third argument) having a certain node (first argument) for scopal top from the list of immediate outscoping statements (second argument).

```
make_q1_node(_,[],_).

make_q1_node(Node,
             [body(Node,Body)|_],
             Node:q1n(global)):-
    Body==global,
    !.

make_q1_node(Node,
             [body(Node,Body)|Scopal_info],
             Node:q1n(Body:_)):-
    !,
    make_q1_node(Node,
                 Scopal_info,
                 Node:q1n(Body:_)).

make_q1_node(Node,
```

```
              [_|Scopal_info],
              Node:q1n(Body)):-
    !,
    make_q1_node(Node,
                 Scopal_info,
                 Node:q1n(Body)).
```

make_q2_node/3

The procedure `make_q2_node/3` collects the information pertaining to a two-place quantifier subtree (third argument) having a certain node (first argument) for scopal top from the list of immediate outscoping statements (second argument).

```
make_q2_node(_,[],_).

make_q2_node(Node,
              [body(Node,Body)|Scopal_info],
              Node:q2n(Restriction,Body:_)):-
    !,
    make_q2_node(Node,
                 Scopal_info,
                 Node:q2n(Restriction,Body:_)).

make_q2_node(Node,
              [restr(Node,Restriction)|Scopal_info],
              Node:q2n(Restriction:_,Body)):-
    !,
    make_q2_node(Node,
                 Scopal_info,
                 Node:q2n(Restriction:_,Body)).

make_q2_node(Node,
              [_|Scopal_info],
```

```
                Node:q2n(Restriction,Body)):-
    !,
    make_q2_node(Node,
                 Scopal_info,
                 Node:q2n(Restriction,Body)).
```

make_paratactic_node/3

The procedure make_paratactic_node/3 collects the information pertaining to a subtree (third argument) having a paratactic predicate node (first argument) for scopal top from the list of immediate outscoping statements (second argument).

```
make_paratactic_node(Node,[ptop(Node,Para)|_],
                 Node:ptop(Para:_)):-
    !.

make_paratactic_node(Node,
                 [_|Scopal_info],
                 Node:ptop(Paratactic_argument)):-
    !,
    make_paratactic_node(Node,
                   Scopal_info,
                   Node:ptop(Paratactic_argument)).
```

separate_tds_statements/4

The procedure separate_tds_statements/4 separates a list of TDS statements into different categories. Lexical statements (lx/3-terms), immediate outscoping statements (having restr/2, body/2, or ptop/2 for main functor), and coindexation statements (having c1/2 or c2/2 for main functor) are separated.

```
separate_tds_statements([],[],[],[]).

separate_tds_statements([v_sf(_,_)|Tail],
```

```
                        Lexical_info,
                        Scopal_info,
                        Coindexations):-
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([lx(A,B,C)|Tail],
                        [lx(A,B,C)|Lexical_info],
                        Scopal_info,
                        Coindexations):-
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([ptop(A,B)|Tail],
                        Lexical_info,
                        [ptop(A,B)|Scopal_info],
                        Coindexations):-
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([body(A,B)|Tail],
                        Lexical_info,
                        [body(A,B)|Scopal_info],
                        Coindexations):-
```

```
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([restr(A,B)|Tail],
                        Lexical_info,
                        [restr(A,B)|Scopal_info],
                        Coindexations):-
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([c1(A,B)|Tail],
                        Lexical_info,
                        Scopal_info,
                        [c1(A,B)|
                           Coindexations]):-
    !,
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).

separate_tds_statements([c2(A,B)|Tail],
                        Lexical_info,
                        Scopal_info,
                        [c2(A,B)|
                           Coindexations]):-
    !,
```

```
    separate_tds_statements(Tail,
                            Lexical_info,
                            Scopal_info,
                            Coindexations).
```

assemble_tree/3

The procedure `assemble_tree/3` assembles a tree (first argument) from a list of nodes (second argument) keeping track of the remaining nodes not put into the tree (third argument). This procedure is only sensitive to the number of subtrees that are appropriate for each node.

```
assemble_tree(Node:[],
              Nodes_in,
              Nodes_left):-
    select(Node:[],
           Nodes_in,
           Nodes_left).
```

Builds a one leaf tree.

```
assemble_tree(Node:q1n(Body),
              Nodes_in,
              Nodes_left2):-
    select(Node:q1n(Body),
           Nodes_in,
           Nodes_left1),
    assemble_tree(Body,
                  Nodes_left1,
                  Nodes_left2).
```

Builds a tree from a one-place quantifier and its body.

```
assemble_tree(Node:q2n(Restriction,Body),
              Nodes_in,
```

```
                Nodes_left3):-
    select(Node:q2n(Restriction,Body),
           Nodes_in,
           Nodes_left1),
    assemble_tree(Restriction,
                  Nodes_left1,
                  Nodes_left2),
    assemble_tree(Body,
                  Nodes_left2,
                  Nodes_left3).
```

Builds a tree from a two-place quantifier and its restriction and body.

```
assemble_tree(Node:ptop(Paratactic_argument),
              Nodes_in,
              Nodes_left2):-
    select(Node:ptop(Paratactic_argument),
           Nodes_in,
           Nodes_left1),
    assemble_tree(Paratactic_argument,
                  Nodes_left1,
                  Nodes_left2).
```

Builds a tree from a paratactic predicate token with its paratactic argument tree.

check_qeq_statements/2

The predicate `check_qeq_statements/2` succeeds if a list of `qeq` statements is true of a given tree.

```
check_qeq_statements([],_).
```

```
check_qeq_statements([Node1 qeq Node2|Qeqs],
                     Tree):-
    !,
    qeq(Node1,Node2,Tree),
    check_qeq_statements(Qeqs,Tree).
```

qeq/3

A `qeq(`*Node1*`,`*Node2*`,`*Tree*`)` call succeeds if and only if *Node1* is `qeq` *Node2* in the scopal tree *Tree*. It is thus a Prolog definition of the relation of equality modulo quantifiers, as adapted to the current representation of scopal trees.

```
qeq(Node,Node,_):-
    !.

qeq(Node1,Node2,Node1:q2n(_,Node_aux:Body)):-
    !,
    qeq(Node_aux,Node2,Node_aux:Body).

qeq(Node1,Node2,Node1:q1n(Node_aux:Body)):-
    !,
    qeq(Node_aux,Node2,Node_aux:Body).

qeq(Node1,Node2,Tree):-
    !,
    immediate_subtree(Tree,Subtree),
    qeq(Node1,Node2,Subtree).
```

immediate_subtree/2

The predicate `immediate_subtree/2` defines a notion of immediate subtree as it apples to scopal trees as implemented here. The second argument is an immediate subtree of the first argument.

```
immediate_subtree(_:q2n(Subtree,_),Subtree).
```

```
immediate_subtree(_:q2n(_,Subtree),Subtree).

immediate_subtree(_:q1n(Subtree),Subtree).

immediate_subtree(_:ptop(Subtree),Subtree).
```

check_binding_requirement/3

The predicate `check_binding_requirement/3` checks that in each case of coindexation between a quantifier and a predicate, the quantifier outscopes the predicate or is of global scope.

```
check_binding_requirement([],_,_).

check_binding_requirement([c1(Node1,Node2)|
                           Coindexations],
                          Tree,
                          Global_quantifiers):-
    outscopes(Node1,
              Node2,
              Tree,
              Global_quantifiers),
    check_binding_requirement(Coindexations,
                              Tree,
                              Global_quantifiers).

check_binding_requirement([c2(Node1,Node2)|
                           Coindexations],
              Tree,Global_quantifiers):-
    outscopes(Node1,
              Node2,
              Tree,
              Global_quantifiers),
```

```
    check_binding_requirement(Coindexations,
                              Tree,
                              Global_quantifiers).
```

outscopes/3
`outscopes(`*Node1*`,`*Node2*`,`*Tree*`,`*Global_quantifiers*`)` holds if and only if *Node1* outscopes *Node2* in the scopal tree *Tree*, given that *Global_quantifiers* is the list of quantifiers with global scope.

```
outscopes(Node1,_,_,Global_quantifiers):-
    memberchk(Node1,Global_quantifiers),
    !.
```

A global quantifier outscopes any node.

```
outscopes(Node1,Node2,Tree,_):-
    subtree(Tree,Node1,Subtree),
    tree_contains_node(Subtree,Node2).
```

subtree/3
`subtree(`*Tree*`,`*Node*`,`*Subtree*`)` holds if and only if *Subtree* is a subtree of the tree *Tree* and the top node of *Subtree* is *Node*.

```
subtree(Node:X,Node,Node:X):-
    !.

subtree(_:q2n(Restriction,_),Node1,Subtree):-
    subtree(Restriction,Node1,Subtree).

subtree(_:q2n(_,Body),Node1,Subtree):-
    subtree(Body,Node1,Subtree).

subtree(_:q1n(Body),Node1,Subtree):-
    subtree(Body,Node1,Subtree).
```

```
subtree(_:ptop(Paratactic_argument),
        Node1,
        Subtree):-
    subtree(Paratactic_argument,
            Node1,
            Subtree).
```

tree_contains_node/3

`tree_contains_node(`*Tree*`,`*Node*`)` holds if and only if the tree *Tree* includes *Node* as one of its nodes.

```
tree_contains_node(Tree:_,Tree).

tree_contains_node(_:q2n(Subtree,_),Tree):-
    tree_contains_node(Subtree,Tree).

tree_contains_node(_:q2n(_,Subtree),Tree):-
    tree_contains_node(Subtree,Tree).

tree_contains_node(_:q1n(Subtree),Tree):-
    tree_contains_node(Subtree,Tree).

tree_contains_node(_:ptop(Subtree),Tree):-
    tree_contains_node(Subtree,Tree).
```

make_FSs_for_readings/2

The predicate `make_FSs_for_readings/2` turns the list of `reading/3` terms collected in `build_readings/3` into a list of FSs. It also collects the additional information about paratactic argument tokens, i.e. about their shape (only their top nodes being defined directly in the scopal trees) and their anchoring.

```
make_FSs_for_readings([],[]).
```

```
make_FSs_for_readings([reading(Tds_list,
                               Qeqs,
                               Tree)|
                          Reading_triples],
                      [FS|Reading_FSs]):-
    !,
    collect_paratactic_argument_info(
                   Tds_list,
                   Tree,
                   Paratactic_info),
    append(Tds_list,
           Tds_list_open_tail,
           Tds_difference_list),
    path_value(FS,
               cont:tds,
               prolog_term(Tds_difference_list-
                           Tds_list_open_tail,_)),
    append(Qeqs,
           Qeqs_open_tail,
           Qeqs_difference_list),
    path_value(FS,
               cont:qeqs,
               prolog_term(Qeqs_difference_list-
                           Qeqs_open_tail,_)),
    path_value(FS,cont:tree,
               prolog_term(Tree,_)),
    path_value(FS,cont:para,
               prolog_term(Paratactic_info,_)),
    make_FSs_for_readings(Reading_triples,
                          Reading_FSs).
```

collect_paratactic_argument_info/3

The procedure collect_paratactic_argument_info/3 collects additional information about each paratactic argument token, i.e. about its shape (only its top node being defined directly in the scopal tree) and its anchoring. The information about each paratactic predicate token is first stored in an argument_info/3 term and then split into the two-place parg and anchor terms of TDS.

```
collect_paratactic_argument_info(Tds_list,
                                 Tree,
                                 Info_pairs):-
    findall(argument_info(Node,
                          Argument_token,
                          Anchor),
            (subtree_paratactic(Tree,
                                Node:
                                 ptop(Top_node)),
                paratactic_argument_token(
                             Top_node,
                             Argument_token),
                anchoring(Tds_list,
                          Top_node,
                          Anchor)),
            Argument_info_triples),
    split_argument_info(Argument_info_triples,
                        Info_pairs).
```

subtree_paratactic/2

subtree_paratactic(*Tree*,*Subtree*) holds if and only if *Subtree* is a subtree of the tree *Tree* and the top node of *Subtree* dominates the rest of *Subtree* by means of the ptop (paratactic argument top node) relation.

```
subtree_paratactic(Node:ptop(Top_node),
```

```
                    Node:ptop(Top_node)):-
    !.

subtree_paratactic(_:q2n(Restriction,_),Subtree):-
    subtree_paratactic(Restriction,Subtree).

subtree_paratactic(_:q2n(_,Body),Subtree):-
    subtree_paratactic(Body,Subtree).

subtree_paratactic(_:q1n(Body),Subtree):-
    subtree_paratactic(Body,Subtree).

subtree_paratactic(_:ptop(Paratactic_argument),
                    Subtree):-
    subtree_paratactic(Paratactic_argument,
                        Subtree).
```

paratactic_argument_token/2

The procedure `paratactic_argument_token/2` puts together the complex token formed by a token and the tokens which it outscopes in a given tree.

```
paratactic_argument_token(Node:[],Node).

paratactic_argument_token(
            Node:q2n(Tree1,Tree2),
            Node+Tree_token1+Tree_token2):-
    paratactic_argument_token(Tree1,Tree_token1),
    paratactic_argument_token(Tree2,Tree_token2).

paratactic_argument_token(Node:q1n(Tree),
                            Node+Tree_token):-
    paratactic_argument_token(Tree,Tree_token).
```

```
paratactic_argument_token(Node:ptop(Tree),
                          Node+Tree_token):-
    paratactic_argument_token(Tree,Tree_token).
```

anchoring/3

The procedure `anchoring/3` compiles a list of anchoring triples (third argument) from a list of TDS statements (first argument) and a given scopal subtree (second argument). For each case in which a quantifier node binds a predicate node, and the quantifier node lies outside of the given (paratactic argument) subtree, and the predicate node is part of the subtree, it adds the appropriate anchoring triple to the list of anchoring triples. The procedure `anchoring/3` thus implements the anchoring requirement.

```
anchoring([],_,[]).

anchoring([c1(Quantifier_node,
              Predicate_node)|Tds_list],
          Subtree,
          [t(Predicate_node,
             1,
             'Ind'(Predicate_node,1))|Anchor]):-
    binds_from_outside(Quantifier_node,
                       Predicate_node,
                       Subtree),
    !,
    anchoring(Tds_list,Subtree,Anchor).

anchoring([c2(Quantifier_node,
              Predicate_node)|Tds_list],
          Subtree,
          [t(Predicate_node,
```

```
            2,
            'Ind'(Predicate_node,2))|Anchor]):-
    binds_from_outside(Quantifier_node,
                       Predicate_node,
                       Subtree),
    !,
    anchoring(Tds_list,Subtree,Anchor).

anchoring([_|Tds_list],Subtree,Anchor):-
    !,
    anchoring(Tds_list,Subtree,Anchor).
```

quantifier_binds_node_from_outside/3

`quantifier_binds_node_from_outside(`*N1*`,`*N2*`,`*Subtree*`)` holds in case the (quantifier) node, *N1*, lies outside of the given (paratactic argument) subtree, *Subtree*, and the (predicate) node, *N2*, is part of the subtree.

```
binds_from_outside(Node1,Node2,Subtree):-
    \+tree_contains_node(Subtree,Node1),
    tree_contains_node(Subtree,Node2).
```

split_argument_info/2

The information about paratactic predicate tokens is first stored in `argument_info/3` terms. The procedure `split_argument_info/2` splits these terms into two-place `parg` and `anchor` terms of TDS.

```
split_argument_info([],[]).

split_argument_info([argument_info(
          Node,
          Argument_token,
          Anchor)|Tail1],
      [parg(Node,Argument_token),
```

```
        anchor(Node,Anchor)|Tail2]):-
    split_argument_info(Tail1,Tail2).
```

dsp_prolog_term/4

dsp_prolog_term/4 is the main predicate for displaying Prolog terms. It is called from inside the PETFSG system. The first argument is the Prolog term to be displayed. The second one gives the current indentation as a natural number giving the distance in character positions from the left. (This is needed in sty=txt display of FSs.) The third and fourth arguments, also natural numbers, are used to provide numbered and easy to read symbols for the Prolog variables. The third argument is the input counter value and the fourth one is the output counter value, which will be the input counter value to the subsequent dsp_prolog_term/4 call for the same grammar item. (Also see the PETFSG documentation.)

```
dsp_prolog_term(Term,
                Indentation,
                Variable_number_in,
                Variable_number_out):-
    instantiate_prolog_variables(
                   Term,
                   Variable_number_in,
                   Variable_number_out),
    dsp_prolog_term_aux(Term,Indentation).
```

dsp_prolog_term_aux/2

The procedure dsp_prolog_term_aux/2 displays Prolog terms after the cosmetic instantiation of the variables have taken place. The first argument is the Prolog term to be displayed. The second one defines the number of blanks to be produced for indentation.

```
dsp_prolog_term_aux(Term,_):-
    var(Term),
```

```
    !,
    write(Term).

dsp_prolog_term_aux(Node:Tree,Indentation):-
    !,
    dsp_scopal_tree(Node:Tree,Indentation,no).
```

This case finds `cont:tree` values, which are displayed by means of the predicate `dsp_scopal_tree/2`.

```
dsp_prolog_term_aux(DL1-DL2,Indentation):-
    !,
    dsp_difference_list(DL1,DL2,Indentation).

dsp_prolog_term_aux([Head|Tail],Indentation):-
    dsp_list([Head|Tail],Indentation).

dsp_prolog_term_aux(Term,_):-
    write(Term).

dsp_list([Head|Tail],Indentation):-
    !,
    write('['),
    write(Head),
    write(','),
    nl,
    Indentation1 is Indentation + 1,
    dsp_list_aux(Tail,Indentation1).

dsp_list_aux([Head],Indentation):-
    !,
    write_spaces(Indentation),
    write(Head),
```

```
    write(']').

dsp_list_aux([Head|Tail],Indentation):-
    !,
    write_spaces(Indentation),
    write(Head),
    write(','),
    nl,
    dsp_list_aux(Tail,Indentation).
```

dsp_difference_list/3

dsp_difference_list/3 displays difference lists simply by printing each of their elements on a separate line. This is for the difference lists of TDS statements.

```
dsp_difference_list([Head|Tail],DL,Indentation):-
    !,
    write('['), write(Head),
    Indentation1 is Indentation + 1,
    dsp_difference_list_aux(Tail,
                            DL,
                            Indentation1).

dsp_difference_list(DL1,DL2,_):- %% For atoms
    !,
    write(DL1-DL2).
```

dsp_difference_list_aux/3

The procedure dsp_difference_list_aux/3 is a help predicate to the previous one.

```
dsp_difference_list_aux(Tail,DL2,_):-
    (var(Tail); atom(Tail)),
    !,
```

```
    write('|'),
    write(Tail),
    write(']-'),
    write(DL2).
```

For an open (or closed) tail.

```
dsp_difference_list_aux([Head|Tail],
                        DL,
                        Indentation):-
    !,
    write(','),
    nl,
    write_spaces(Indentation),
    write(Head),
    dsp_difference_list_aux(Tail,DL,Indentation).
```

dsp_scopal_tree/3

The procedure `dsp_scopal_tree/3` displays Prolog terms representing scopal trees, with a suitable indentation. The first argument is the scopal tree to be displayed. The second one defines the number of blanks to be produced for indentation. Each subtree starts on a new line. The third argument, a `yes` or a `no`, indicates whether the procedure should produce indentation for the current line. It should not for the first line of the tree.

```
dsp_scopal_tree(Node:[],Indent,To_be_indented):-
    !,
    write_spaces(To_be_indented,Indent),
    write(Node:[]).

dsp_scopal_tree(Node:q2n(Restriction,Body),
                Indent,
                To_be_indented):-
```

```
    !,
    write_spaces(To_be_indented,
                 Indent),
    write(Node),
    write(':q2n('),
    nl,
    Indent2 is Indent + 2,
    dsp_scopal_tree(Restriction,Indent2,yes),
    write(','),
    nl,
    dsp_scopal_tree(Body,Indent2,yes),
    write(')').

dsp_scopal_tree(Node:q1n(Body),
                Indent,
                To_be_indented):-
    !,
    write_spaces(To_be_indented,Indent),
    write(Node),
    write(':q1n('),
    nl,
    Indent2 is Indent + 2,
    dsp_scopal_tree(Body,Indent2,yes),
    write(')').

dsp_scopal_tree(Node:ptop(Body),
                Indent,
                To_be_indented):-
    !,
    write_spaces(To_be_indented,Indent),
    write(Node),
    write(':ptop('),
    nl,
```

```
    Indent2 is Indent + 2,
    dsp_scopal_tree(Body,Indent2,yes),
    write(')').

dsp_scopal_tree(Term,
                Indentation,
                To_be_indented):-
    !,
    write_spaces(To_be_indented,Indentation),
    write('*'), %% Error case.
    write(Term).
```

instantiate_prolog_variables/3

The procedure `instantiate_prolog_variables/3` instantiates Prolog variables with numbered constants, intended to provide a suitable graphical appearance for the variables. The first argument is the Prolog term whose constituent variables are being instantiated. The two other arguments are counters providing index numbers for the variables/constants.

```
instantiate_prolog_variables(Term,
                             Number_in,
                             Number_out):-
    var(Term),
    !,
    make_atom_for_variable(Term,
                           Number_in,
                           Number_out).

instantiate_prolog_variables(Term,
                             Number,
                             Number):-
    simple(Term),
    !.
```

```
instantiate_prolog_variables([Head|Tail],
                             Number_in,
                             Number_out):-
    !,
    instantiate_prolog_variables(Head,
                                 Number_in,
                                 Number_aux),
    instantiate_prolog_variables(Tail,
                                 Number_aux,
                                 Number_out).

instantiate_prolog_variables(Term,
                             Number_in,
                             Number_out):-
    !,
    Term =.. List,
    instantiate_prolog_variables(List,
                                 Number_in,
                                 Number_out).
```

make_atom_for_variable/3

The procedure `make_atom_for_variable/3` makes atoms that provide the textual shape for Prolog variables in printing. Its first argument is a variable to be instantiated. The result may e.g. be `make_atom_for_variable('XX5',5,6).`

```
make_atom_for_variable(Atom_for_variable,
                       Variable_number_in,
                       Variable_number_out):-
    name(Variable_number_in,Number_name),
    name(Atom_for_variable,[88,88|Number_name]),
    Variable_number_out is Variable_number_in + 1.
```

Some simple printing procedures.

```
write_spaces(0):-
    !.

write_spaces(N):-
    !,
    write(' '),
    M is N -1,
    write_spaces(M).

write_spaces(yes,N):-
    !,
    write_spaces(N).

write_spaces(no,_):-
    !.

call_user_petfsg_act(_,_). %% Not used here.
```

•

Appendix D: The Types

This is the type system of the present grammar, with short explanations of the purpose of each type. The **IST** column gives the *immediate supertype*.

Type	Description	IST
aops	absent or present sign	
neg_s	absent sign	aops
sign	sign (‘positive’)	aops
lexitem	lexical sign	sign
lxm	lexeme	lexitem
infl_lxm	lexeme subject to inflection	lxm
word	surface word form	lexitem
punct_mark	punctuation mark	word
word_valency	word carrying complement valency	word
word_no_valency	word lacking complement valency	word

Type	**Description**	**IST**
phrase	phrase	sign
phrase_wp	phrase from word, not taking any complement	phrase
phrase_hc	head complement phrase	phrase
phrase_ssh	subject specifier head phrase	phrase
phrase_dsh	determiner specifier head phrase	phrase
phrase_mh	modifier head phrase	phrase
phrase_hp	head punctuation mark construction	phrase
head_sort	topmost head value type	
verb	'verbal'	head_sort
v	ordinary part-of-speech verb	verb
cplzer	complementizer	verb
agrpos	carrying agreement information	head_sort
determin	determiner	agrpos
nom	'nominal'	agrpos
cnnoun	common noun	nom
perspron	personal pronoun	nom
prnoun	proper noun	nom
adj	adjective	head_sort
adv	adverb	head_sort
conj	conjunction	head_sort
prep	preposition	head_sort
mk_prep	marker preposition	prep
pred_prep	predicative preposition	prep

Type	**Description**	**IST**
agr_sort	agreement value	
cont_sort	semantic substructure	
holes	hole collection	
holes_v	hole collection for a verbal expression	holes
holes_d	hole collection for a determiner expression	holes
hole_type	hole topmost type	
closed	‘closed’ hole	hole_type
hole	‘open’ hole	hole_type
hole_n	hole for verbal connection	hole
hole_v	hole for nominal connection	hole
boolean	boolean value	
no	false	boolean
yes	true	boolean
number_sort		
sing	singular	number_sort
plur	plural	number_sort
def_sort	definiteness value	
def	definite	def_sort
indef	indefinite	def_sort
case_sort		
nomin	nominative, or rather, non-genitive	case_sort
subc	subject case	nomin
objc	object case	nomin
gen	genitive	case_sort

Type	**Description**	**IST**
`pers_sort`	grammatical person value	
`p1`	first person	`pers_sort`
`p2`	second person	`pers_sort`
`p3`	third person	`pers_sort`
`vform_sort`	verbal inflection	
`finite`	finite	`vform_sort`
`present`	present tense	`finite`
`preterit`	preterit	`finite`
`infinite`	infinite	`vform_sort`
`infinitive`	infinitive	`infinite`
`prp`	present participle	`infinite`
`psp`	past participle	`infinite`
`v_kind_sort`	sort for verbal kinds	
`aux`	auxiliary verb	`v_kind_sort`
`notaux`	not an auxiliary verb	`v_kind_sort`

Appendix E: Downloadable TDS Grammar Files

These files have been used to create the PETFSG-II application described in this document. They are found at URL `http://stp.ling.uu.se/~matsd/tds/jos/f/`.

`gram_tds.pl`: This file contains the main parts of the TDS grammar (Appendix B).

`application_procedures.pl`: This file contains the Prolog procedures used by the TDS grammar (Appendix C).

`lex_tds.pl`: This file contains additional lexical entries.

`suites_tds.pl`: This file defines a few test suites.

`make_tds.pl`: This file contains the Prolog calls used to create the TDS grammar saved state (under SICSTUS Prolog).

RUUL, Reports from Uppsala University, Dept. of Linguistics,
Box 527, SE-751 20 UPPSALA, Sweden.
`http://www.ling.uu.se/ruul/`
This volume, nr 36 (Januari 2003), contains
Two Reports on Computational Syntax and Semantics
by Mats Dahllöf.
ISBN: 91-973737-2-9 (this volume). ISSN: 0280-1337 (*RUUL*).

Printed by Universitetstryckeriet, Uppsala.