An implementation of
token dependency semantics
using the PETFSG grammar tool

Mats Dahllöf
Department of Linguistics, Uppsala University
E-mail: mats.dahllof@ling.uu.se
January 7, 2002

Work in progress! Demo, updates, and related material:
http://stp.ling.uu.se/~matsd/tds/1/

1 Introduction

This report\(^1\) describes an implementation of an HPSG-style formal grammar and semantics, which will be called ‘token dependency semantics’ (henceforth TDS), version 1. The theoretical background is given in my article ‘Token dependency semantics and the paratactic analysis of intensional constructions’ (available from the author). The relation between the presentations is fairly direct. The present report gives a technical account of the implemented grammar, whose linguistic aspects are explained and defended in the other article. The main purpose of the present document is 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 grammar as implemented here uses the system and formalism ‘Prolog-embedding typed feature structure grammar’ (henceforth PETFSG) version 3. (See http://stp.ling.uu.se/~matsd/petfsg/3/ for documentation.) This system uses typed feature structures in which Prolog-terms may be embedded. Prolog procedures may be called from inside the rules. Here, Prolog-terms are used to represent semantic objects. Prolog is only used to append lists and to create token identifiers. Otherwise, the grammar relies on unification.

\(^1\)A report on a previous version of this work was given in RUUL (Reports from Uppsala University Dept of Linguistics) 34, 1999.
many cases) are underspecified with respect to scopal relations. Additional Pro-
log procedures, external to the grammar, are used to compute the possible scopal
resolutions. These are applied as ‘postparse operations’, see below.

The files of the implementation are available at the URL
http://stp.ling.uu.se/~matsd/tds/3/f/. An overview is given in
appendix D.

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 below.)

A sign FS is associated with features as follows:

    sign features
    [token:prolog_term,
      head:head_sort,
      spr:list,
      spr_inc:boolean,
      comp:comp_list,
      slash:list,
      cont:cont_sort,
      pm:punct_mark]

The features are used as described in the background article, with the exception of
spr_inc, slash, and pm. The boolean feature spr_inc means ‘specifier incorporated’ and is used to distinguish phrases which contain a specifier con-
stituent from those that do not (in cases where spr is the empty list). spr_inc plays a role in the treatment of adjectives. 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 cont(ent) features

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,
The features hook and tds (token dependency statements), are used as described in the background article. The hole features are collected under the feature h (see below). The features glob(al), tree, and rss (reading-specific statements) are used in the computation of scopally resolved readings (see below).

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_v structures carry a mod_hole and a
spec_hole. The comp_holes feature carry a list value. Its elements are hole
objects, each associated with the corresponding element on the comp list.

2 Tense, perfectivity, and progressivity

The grammar only ‘records’ tense, perfectivity, and progressivity. The treatment
presented should be replaced by a better one.

The tense, perfectivity, and progressivity information is associated with a
verbal head feature vs_f (verbal semantic features). Its value is a complex
term ‘[T, P, Q]’ containing three features, T for basic tense, preterit being the
‘+’ case. Three possible values: present (‘-’), preterit (‘+’), and tenseless (‘n’,
for the case of infinitive). The P value represents perfectivity, non-perfect (‘-’)
and perfect (‘+’) being the alternatives. Finally, Q stands for progressivity. Non-
progressive (‘-’) and progressive (‘+’) are the possibilities here. Examples:

<table>
<thead>
<tr>
<th></th>
<th>pres, non-perf, non-progr</th>
<th>Mary smiles.</th>
</tr>
</thead>
<tbody>
<tr>
<td>‘[-, -, -]’</td>
<td>pres, non-perf, progr</td>
<td>Mary is smiling.</td>
</tr>
<tr>
<td>‘[-, -, +]’</td>
<td>pres, perf, non-progr</td>
<td>Mary has smiled.</td>
</tr>
<tr>
<td>‘[-, +, -]’</td>
<td>pres, perf, progr</td>
<td>Mary has been smiling.</td>
</tr>
<tr>
<td>‘[-, +, +]’</td>
<td>pret, non-perf, non-progr</td>
<td>Mary smiled.</td>
</tr>
<tr>
<td>‘[+,-,-,-]’</td>
<td>pret, non-perf, progr</td>
<td>Mary was smiling.</td>
</tr>
<tr>
<td>‘[+,-,-]’</td>
<td>pret, perf, non-progr</td>
<td>Mary had smiled.</td>
</tr>
<tr>
<td>‘[+,+,-,-]’</td>
<td>pret, perf, progr</td>
<td>Mary had been smiling.</td>
</tr>
</tbody>
</table>

These triples are transferred to the verb EPS, connected to the feature s_f. The
three features are consequently seen as properties of the main verb token.

3 Token identifier assignment

The TDS grammar presupposes that the grammatical descriptions uniquely iden-
tify the tokens to which they apply. This means that each token must be as-
signed 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:
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’ followed by the position number, an underscore character, and the orthographic form, e.g. ‘t3_house’ if the token is an instance of `house` at position 3.

### 4 Inflectional morphology

The following Prolog clauses define the (very simple) morphological processes used by the grammar:

```prolog
morphology(id,X,X).

morphology(affx(A),X,Y):-
    name(X,N),
    name(A,AN),
    append(N,AN,NN),
    name(Y,NN).
```

The identifier ‘id’ represents identity and ‘affx(A)’ addition of the affix A.

### 5 The representation of TDS information

The TDS relations appear as follows:

‘c1’: Coindexation between a quantifier and a predicate with respect to the first argument position.

‘c2’: Coindexation between a quantifier and a predicate with respect to the second argument position.

‘restr’: Immediate outscoping between a quantifier and its restriction.

‘body’: Immediate outscoping between a quantifier and its body.
‘ptop’: Immediate outscoping between a paratactic predicate and the scopally topmost non-quantifier node of the paratactic argument.

‘ptop’: The relation equality modulo quantifiers, due to Copestake et al. (2001). It makes it possible to state scope constraints (of lexical origin).

‘L’: This relation is three-place here. It holds of a token, the lexical expression of which it is an instance, and a term describing its logical valency, see below.

‘parg’: Holds of a paratactic relation and the whole paratactic argument subtree.

‘anchor’: Holds of a paratactic relation and its anchor argument.

‘Ind’: Is used in such a way that ‘Ind (N, T)’ is the individual associated with argument position N of the token T given current bindings.

Semantic valencies are represented by terms as follows:

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

‘q2’: Two-place quantifier (i.e. two).

‘p([a1])’: Intransitive non-paratactic relation (i.e. smile).

‘p([a1,a2])’: Transitive non-paratactic relation (i.e. read).

‘p([para])’: Intransitive paratactic relation (i.e. probably).

‘p([a1,para])’: Transitive relation with a second paratactic argument position (i.e. say).

6 Postparse operations

The postparse alternatives make partial selections of information to display and compute scopal resolutions from underspecified semantic representations. The following are defined:

- ‘nomod’: No postparse operation is applied.
- ‘cont’: Only the value of the feature cont is selected for output.
- ‘resl1’: The set of scopal resolutions of the cont value are computed. This version assumes that names take widest possible scope. Only the value of the feature cont is then selected for output.
- ‘resl2’: The ‘resl2’ resolution is different from ‘resl1’ resolution only in allowing names to be outscoped by other quantifiers. Only the value of the feature cont is then selected for output.
6.1 Specification of readings

The content features `glob(al)`, `rss`, and `tree` are used in the specification of scopally resolved readings. The feature `glob(al)` is a list of the a priori topmost quantifiers in the case of ‘resl1’ resolution. 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 `disp_prolog_term/3` is responsible for displaying these tree structures in a suitable fashion.)

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.

\[
\begin{align*}
[L(t_1_{Mary}, Mary, q1), \\
  L(t_2_{says}, say, p([a1, para])), \\
  c1(t_1_{Mary}, t_2_{says}), \\
  \text{ptop}(t_2_{says}, P0), \\
  P0 \text{ qeq t6_smiled, } \\
  L(t_4_{two}, two, q2), \\
  \text{restr}(t_4_{two}, P1), \\
  P1 \text{ qeq t5_students, } \\
  L(t_5_{students}, student, p([a1])), \\
  c1(t_4_{two}, t_5_{students}), \\
  L(t_6_{smiled}, smile, p([a1])), \\
  c1(t_4_{two}, t_6_{smiled})]
\end{align*}
\]

Here, underspecification is due to the Prolog variables ‘P0’ and ‘P1’ and to the absence of ‘body’ statements. The scope resolution algorithms work by instancing variables and computing additional information.

One of the (two) readings of this example given by the ‘resl1’ resolution algorithm can be characterized as follows:

\[
t_1_{Mary} \text{<<} t_4_{two} \text{<} t_5_{students} \hspace{1cm} t_2_{says} \text{[t6_smiled]}
\]

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

\[
\begin{align*}
[L(t_1_{Mary}, Mary, q1), \\
  L(t_2_{says}, say, p([a1, para])), \\
  c1(t_1_{Mary}, t_2_{says}),
\end{align*}
\]
The \text{glob} \,(al) \,value \,is \,the \,list \,of \,the \,only \,\textit{a \,priori} \,topmost \,quantifier:

$$\text{[t1\_Mary]}$$

The \text{rss \,TDS} \,statements \,define \,a \,quantifier \,body, \,a \,paratactic \,argument, \,and \,an \,anchor:

$$\text{[body(t4\_two,t2\_says),}$$
$$\text{parg(t2\_says,t6\_smiled),}$$
$$\text{anchor(t2\_says,[t(t6\_smiled,1,Ind(t6\_smiled,1))])]}$$

The \text{tree} \,value \,gives \,an \,overview \,of \,the \,scopal \,situation \,and \,is \,printed \,as \,follows:

$$\text{[t4\_two:q2n(t5\_students,t2\_says),}$$
$$\text{[t5\_students:[]],}$$
$$\text{[t2\_says:ptop(t6\_smiled),}$$
$$\text{[t6\_smiled:[]]]]}$$

\section{Phrase structure rules}

A few comments should be made concerning the phrase structure rules. The \textsc{HPSG} head-complement rule is implemented with the help 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 syntactic (nominal) head.

\texttt{phrase\_from\_word}: \,This \,rule \,covers \,the \,case \,when \,the \,head-complement \,rule \,applies \,with \,an \,empty \,list \,of \,complements, \,i.e. \,when \,the \,mother \,phrase \,dominates \,a \,head \,word \,without \,sisters.

8
**hd_comps**: This rule covers the case when the head-complement rule applies with a non-empty \( \text{comps} \) list on the head. Only one complement is incorporated on each application of the rule. The complement holes in the \( \text{comp_holes} \) list correspond one-by-one to the \( \text{comps} \) 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.

**subjSpr_hd**: This rule is for specifier-head constructions in which the specifier is a subject to a verbal (syntactic) head, which is also the semantic head.

**detSpr_hd**: This 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.

**mod_hd**: Premodifier-head rule.

**phr_punct**: Phrase-punctuation-mark rule. A phrase without a punctuation mark—formally, \( \text{pm:neq:p} \)—combines with a punctuation mark and forms a phrase with the feature \( \text{pm} \) specifying the sister punctuation mark. The mother and daughter phrases otherwise carry the same feature values.

In order to make it easier to see which rule has produced an analysis, there is a subtype to the type phrase for each phrase structure rule:

- **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

These distinctions do not play any linguistic role.

**References**


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
    mod neg_s
    vform present
    v_s1 [-,-,-]

spr =>
spr_inc y=>
comps =>
slash =>

cont_sort
    hole_v

hook
    htop t2_says
    exlarg t1_Mary

h
    holes_v
    mod_hole closed
    subj_hole closed
    comp_holes <>

cont
    [L(t1_Mary, Mary, q1),
     L(t2_says, say, p([a1, para])),
     cl(t1_Mary, t2_says),
     ptop(t2_says, E0),
     E0 qeq t6_smiled,
     L(t4_two, two, q2),
     xastx(t4_two, E1),
     E1 qeq t5_students,
     L(t5_students, student, p([a1])),
     cl(t4_two, t5_students),
     L(t6_smiled, smile, p([a1])),
     cl(t4_two, t6_smiled)]
```

The two readings are generated by the ‘res11’ postoperation. The name quantifier is assigned global scope. First reading:
Second reading:
These are a few examples of lexical entries. The verb form says:
The adverb *intentionally*:
The adverb *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 a file with additional lexical entries (‘lex.tds.pl’).

grammar ‘TDS-1’.

### Declaration of types and features

```plaintext
declarion

[aops subsumes [ % absent or present sign
  neg_s subsumes [], % absent sign
  sign subsumes [ % present sign
    lexititem subsumes [ % lexical item
      lxnm subsumes [ % lexeme
        inf1_lxnm subsumes []], % inflected lexeme
      word subsumes [ % word
        word_sat subsumes [], % saturated
        word_unsat subsumes []]], % unsaturated
    phrase subsumes [ % phrase-level signs
      phrase_wp subsumes [], % phrase from word
      phrase_hc subsumes [], % head-complement
      phrase_ssh subsumes [], % subject-head
      phrase_dsh subsumes [], % 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
  nom subsumes [ % nominal
    cnnoun subsumes [], % common nouns
    perspron subsumes [], % personal pronoun
    prnoun subsumes []], % proper nouns
  adj subsumes [], % adjective
  adv subsumes [], % adverb
  conj subsumes [], % conjunction
  determin subsumes [], % determiner
  prep subsumes []], % preposition
  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 [], % "closed" (unavailable)
  hole subsumes [ ] % "open" hole
    hole_n subsumes [], % for nominals
    hole_v subsumes []], % for verbals

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 inflection
  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

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 []]]

where

[sign features
  [token:prolog_term,
   head:head_sort,
   spr:list,
   spr_inc:boolean, % specifier incorporated
   comps:list,
   slash:list,
   cont:cont_sort,
   pm:punct_mark],

lexitem features
  [lexeme:prolog_term],

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

hole features
  [ltop:prolog_term],

hole_n features
  [key:prolog_term],

hole_v features
Information concerning tense, perfectivity, and progressivity is associated with a
verbal head feature \( v_{sf} \) (verbal semantic features). Its value is a complex term
\( \{T, P, Q\} \) containing three features, \( T \) for basic tense, preterit being the ‘+’ case. Three
possible values: present (‘−’), preterit (‘+’), and tenseless (‘n’, for the case of infinitive). \( P \)
represents perfectivity, non-perfect (‘−’) and perfect (‘+’) being the alternatives. Finally, \( Q \)
stands for progressivity. Non-active (‘−’) and active (‘+’) are the possibilities
here. (This is only a tentative treatment of these features.)

**Initial symbol definition**

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

**Leaf symbol definition**
leaf_symbol word.

**The phrase structure rules**

**Phrase from word rule**

phrase_from_word rule

[phrase_wp,
 token: <>T1,
 head:xHFP,
 comps:{},
 spr:xSpr,
 spr_inc:no,
 slash:{},
 cont:xCont,
 pm:neg_p] ===>

[ [word_unsat,
  token: <>T1,
  head:xHFP,
  comps:{},
  spr:xSpr,
  spr_inc:no,
  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 Prolog call computes the mother’s slash and tds values.

hd_comps rule

[xMtr,
 phrase_hc,
 token: <>(T1+T2),
 head:xHFP,
 comps:xComps,
 spr:xSpr,
 spr_inc:xSprInc,
 cont:[hook:xHook,
The specifier-head rule, subject case

subject-spr_head rule

[xMtr,
 phrase_ssh,
 token: <>(T1+T2),
 head:xHFP,
 spr:{},
 spr_inc:yes,
 comp:{},
 cont:[hook:xHook,
    h:[subj_hole:closed,
       comp_holes:xCHs,
       mod_hole:xMH}],
 pm:neg_p]  

[ xSpr,
  token: <>T1, % Spec-principle not needed.
  pm:neg_p]  

[ [xHead,
   token: <>T1,
   head:xHFP,
   spr:xSpr,
   spr_inc:xSprInc,
   comp:xComp^xComps,
   cont:[hook:xHook,
      h:[subj_hole:xSH,
         comp_holes:xCH^xCHs,
         mod_hole:xMH]],
   pm:neg_p],
 [xComp,
   token: <>T2,
   cont:[hook:xCH],
   pm:neg_p],

prolog [mother_daughters,
   xMtr,xHead,xComp].
The specifier-head rule, determiner case

Pollard’s and Sag’s (1994: 51) ‘Spec Principle’ is hard-coded into the rule.

det_spr_hd rule

[ xMtr,
  phrase_dash,
  token: <>(T1+T2),
  head:xHFP,
  spr:{},
  spr_inc:yes,
  comps:{},
  cont:[hook:xHook,
        h:[spec_hole:closed,
           mod_hole:xMH]],
  pm:neg_p] ===>

[ [xSpr,
   token: <>T1,
   head:[determiner,
         spec:xHead],
   cont:[hook:xHook,
         h:[spec_hole:xSpecHook,
            mod_hole:xMH]],
   pm:neg_p],


21
The premodifier-head rule

mod_hd rule

[xMtr,
 phrase_mh,
 token: <>(T1+T2),
 head:xHFP,
 spr:xSpr,
 spr_inc:xSprInc,
 comps:xComps,
 cont:[hook:xModHook,
       h:[subj_hole:xSH,
           comp_holes:xCHs,
           mod_hole:xMH]],
 pm:neg_p]

  ===>

  [ [xMod,
      token: <>T1,
      head:[mod:xHead,
            premod:yes],
      spr:{},
      comps:{},
      cont:[hook:xModHook,
            h:[mod_hole:xHdHook]],
      pm:neg_p],

  [xHead,
   token: <>T2,
   head:xHFP,
   spr:xSpr,
   spr_inc:xSprInc,
comps:xComps,
cont:[hook:xHdHook,
    h:[subj-hole:xSH,
        comp_holes:xCHs,
        mod_hole:xMH],
    pm: neg_p],

prolog [mother_daughters,
    xMtr, xMod, xHead]].

Phrase-punctuation-mark 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:{},
    spr_inc:xSprInc,
    comps:{},
    cont:xCont,
    pm:xPM] ===>

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

    [xPM,
    punct_mark]].

Punctuation mark entries

'.' >>>
    full_stop.

', ' >>>
Lexical information

General definitions

content_lexitem is_short_for
  [token: <>Token,
   cont:[hook:[ltop: <>Token]],
   slash:{},
   spr_inc:no,
   pm:neg_p].

'content_lexitem' applies to lexical entries which contribute to semantics. The features token and cont:hook:ltop are defined.

form_lexitem is_short_for
  [slash:{},
   spr_inc:no,
   pm:neg_p].

'form_lexitem' applies to lexical entries which do not contribute to semantics. The features token and cont:hook:ltop are not defined.

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

Subject form NPs.
np_obj is_short_for
[head:[nom,
    case:objc],
  spr:{},
  comps:{},
  slash:{},
  pm:neg_p].

Object form NPs.

sentence is_short_for
[head:[v,
    vform:finite],
  spr:{},
  spr_inc:yes,
  comps:{}].

Definition of sentence.

inf_phrase_plain 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.

\[
\text{no_mod is_short_for} \\
\text{[head:[mod:neg_s],} \\
\text{cont:[h:[mod_hole:closed]].} \\
\]

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

### Proper nouns

\[
\text{proper_noun is_short_for} \\
\text{[nfs content_lexitem,} \\
\text{token: <>Token,} \\
\text{lexeme: <>Lexeme,} \\
\text{head:[prn noun,} \\
\text{num:sing,} \\
\text{pers:p3],} \\
\text{spr:{},} \\
\text{comps:{},} \\
\text{nfs no_mod,} \\
\text{cont:[hook:[key: <>Token],} \\
\text{tds: <>["L'(Token,Lexeme,q1)]]].} \\
\]

'Galileo' >>>
\[
\text{[word_unsat,} \\
\text{nfs proper_noun,} \\
\text{lexeme: <>'Galileo'].} \\
\]

### Personal pronouns

\[
\text{personal_pronoun is_short_for} \\
\text{[nfs content_lexitem,} \\
\text{token: <>Token,} \\
\text{lexeme: <>Lexeme,} \\
\text{head:perspron,} \\
\text{spr:{},} \\
\text{comps:{},} \\
\text{nfs no_mod,} \\
\text{cont:[hook:[key: <>Token],} \\
\text{tds: <>["L'(Token,Lexeme,q1)]]].} \\
\]

he >>>
\[
\text{[nfs personal_pronoun,} \\
\text{head:[pers:p3,} \\
\text{}} \\
\]

26
Determiners

determiner is_short_for
[nfs form_lexitem,
  word_sat,
  token: <+>Token,
  lexeme: <+>Lexeme,
  head:determiner,
  spr: {},
  comps: {},
  nfs no_mod,
  cont:[hook:[key: <+>Token],
    h:[spec_hole:[key: <+>Token,
      ltop: <+>LTOP]],
    tds: <+>['L'(Token,Lexeme,q2),
      restr(Token,Restr),
      Restr qeq LTOP]].

many >>>
[nfs determiner,
  head:[spec:[head:[num:plur]]],
  lexeme: <+>‘many’].

the >>>
[nfs determiner,
  head:[spec:[head:[num:sing]]],
  lexeme: <+>‘the_sing’].

the >>>
[nfs determiner,
  head:[spec:[head:[num:plur]]],
  lexeme: <+>‘the_plur’].

Two senses of the, as number is reflected (lexically) in the l0 value.

Noun morphology

lexrule sing_noun
  morph id
  input [infl_lxm,
    token:x0,
lexeme:x1,
head:[x2,
    cnnoun],
spr:x3,

output [word_unsat,
token:x0,
lexeme:x1,
head:[x2,
    cnnoun,
    num:sing],
spr:x3,

lexrule plur_noun
morph affx(s)
input [infl_lxm,
token:x0,
lexeme:x1,
head:[x2,
    cnnoun],
spr:x3,

output [word_unsat,
token:x0,
lexeme:x1,
head:[x2,
    cnnoun,
    num:plur],
spr:x3,
The common kind of singular/plural noun inflection in English.

**Common nouns requiring a specifier**

commmon_noun_spr is_short_for
  [nfs content_lexitem,
   token: <>Token,
   lexeme: <>Lexeme,
   head:[cnnoun,
       pers:p3],
   spr:{[head:determin,
       pm:neg_p]},
   comps:{},
   nfs no_mod,
   cont:[hook:[key: <>SprKey,
       ltop: <>Token],
       h:[subj_hole:closed],
       tds: <>['I' (Token,Lexeme,p([a1])),
       c1(SprKey,Token)]].

This does not give us bare plurals.

book >>>
  [inf_lx_m,
   nfs common_noun_spr,
   lexeme: <>book].

**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:[pers:p3,
       num:sing]]}].

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

preterit_third_singular is_short_for
[head:[vform:present,
    v_sf: <>[‘+’,-’,-’]],
spr:{{head:[pers:p3,
    num: sing]]}}.

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

infinite_form is_short_for
[head:[vform:infinite,
    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 id
  input [infl_lxm,
    token:x0,
    lexeme:x1,
    head:[x2,
      verb],
spr:x3,
spr_inc:x4,
comps:x5,
slash:x6,
cont:x7]
  output [word_unsat,
    nfs infinite_form,
    token:x0,
    lexeme:x1,
    head:x2,
lexrule v_infl_third_plural
morph id
input [infl_lxm,
  token:x0,
  lexeme:x1,
  head:[x2,
    verb],
  spr:x3,
  spr_inc:x4,
  comp:x5,
  slash:x6,
  cont:x7]
output [word_unsat,
  nfs present_third_plural,
  token:x0,
  lexeme:x1,
  head:x2,
  spr:x3,
  spr_inc:x4,
  comp:x5,
  slash:x6,
  cont:x7].

lexrule v_infl_third_singular
morph affx(s)
input [infl_lxm,
  token:x0,
  lexeme:x1,
  head:[x2,
    verb],
  spr:x3,
  spr_inc:x4,
  comp:x5,
  slash:x6,
  cont:x7]
output [word_unsat,
  nfs present_third_singular,
lexrule v_infl_pret
  morph affx_ch_e(ed)
  input [inf1_lxm,
    token:x0,
    lexeme:x1,
    head:[x2,
      verb],
    spr:x3,
    spr_inc:x4,
    comps:x5,
    slash:x6,
    cont:x7]
  output [word_unsat,
    nfs_preteritum,
    token:x0,
    lexeme:x1,
    head:x2,
    spr:x3,
    spr_inc:x4,
    comps:x5,
    slash:x6,
    cont:x7].

lexrule v_infl_psp
  morph affx_ch_e(ed)
  input [inf1_lxm,
    token:x0,
    lexeme:x1,
    head:[x2,
      verb],
    spr:x3,
    spr_inc:x4,
    comps:x5,
    slash:x6,
Intransitive verbs

Verb_intransitive is_short_for
[nfs verb_general,
token: <>Token,
lexeme: <>Lexeme,
spr: {nfs np_subj},
comps: {},
cont: [hook: [extarg: <>SubjKey],}
smile >>>
   [infl_lxm,
    nfs verb_intransitive,
    lexeme: <>smile].

Ordinary transitive verbs

verb_transitive is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    spr:{[nfs np_subj]},
    comps:{[nfs np_obj]},
    cont:{[hook:[extarg: <>SubjKey],
      h:[subj_hole:[key: <>SubjKey],
        comp_holes:{[key: <>ObjKey]},
      tds: <>'L'(Token,Lexeme,p([a1,a2]),
        c1(SubjKey,Token),
        c2(ObjKey,Token))].

The an feature is given the nil value ’-’ to indicate that anchoring is not applicable.

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

Paratactic verbs

verb_paratactic is_short_for
   [nfs verb_general,
    token: <>Token,
    lexeme: <>Lexeme,
    spr:{[nfs np_subj]},
    comps:{[head:verb,
      comp_holes:{},
      spr:{},
      pm:neq_p]},
    cont:{[hook:[extarg: <>SubjKey],
      h:[subj_hole:[key: <>SubjKey],
        comp_holes:[]}].
Raising verbs

verb_subj_raising is_short_for
[nfs verb_general,
token: <>Token,
lexeme: <>Lexeme,
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: <>['L'(Token,Lexeme,p([al,para]),
        ptop(Token,CTT),
        CTT qeq LTOP])].

verb_subj_raising_plain is_short_for
[nfs verb_subj_raising,
comps:{nfs inf_phrase_plain}].

Takes as complement an infinitival phrase without a to complementizer.

would >>>
[word_unsat,
    nfs verb_subj_raising_plain,
    nfs preteritum,
    lexeme: <>would].

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

Takes as complement an infinitival phrase with a to complementizer.
ought >>>
[word_unsat,
  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,
  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: <>['L'(Token,Lexeme,p([a1,para]),
      cl(SubjKey,Token),
      ptop(Token,CTT),CTT qeq LTOP]}].

The verb EP and the complement EP are coindexed with respect to their a1 arguments. Takes as complement an infinitival phrase with a to complementizer.

intend >>>
[inf1 lxm,
  nfs verb_subj_control,
  lexeme: <>intend].

Object raising verbs

verb_obj_raising is_short_for
[nfs verb_general,
  token: <>Token,
  lexeme: <>Lexeme,
  spr:{{nfs np_subj}},
  comps:{{nfs np_obj},
    [nfs inf_phrase_to,
      spr:{{}}]},
  cont:{hook:{extarg: <>SubjKey},
    h:[subj Hole:[key: <>SubjKey],
      comp_holes:{{ltop: <>LTOP,
                   extarg: <>SubjKey}}]}.}
comp_holes:{[key: <>ObjKey],
    [ltop: <>LTOP,
      extarg: <>ObjKey]}],
  tds: <>'L'(Token, Lexeme, p([a1, para]),
    cl(SubjKey, Token),
    ptop(Token, CTT),
    CTT qeq LTOP])}.

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,
    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: <>'L'(Token, Lexeme, p([a1, para]),
          cl(SubjKey, Token),
          c2(ObjKey, Token),
          ptop(Token, CTT),
          CTT qeq LTOP])]}.

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

Auxiliary verbs

verb_aux is_short_for
  [word_unsat,
    nfs form_lexitem,
These verbs only serve to specify 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: &lt;'n',_,_]}]}].

has >>>  
[nfs verb_temp_have,  
nfs_present_third_singular,  
comps:{{head:[v_sf: &lt;'-',_,_]}]}].

have >>>  
[nfs verb_temp_have,  
nfs_present_third_plural,  
comps:{{head:[v_sf: &lt;'-',_,_]}]}].

had >>>  
[nfs verb_temp_have,  
nfs_preteritum,  
comps:{{head:[v_sf: &lt;'+',_,_]}]}].

**Progressive be**

verb_progr_be is_short_for  
[word_unsat,
Progressive be requires a v_kind:notaux complement (cf. *is having seen*).

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_unsat,
nfs content_lexitem,
token: <>Token,
lexeme: <>Lexeme,
head:adj,
spr:{},
comps:{},
head:[mod:[head:cnnoun,
   spr_inc:no,
   comps:{},
   pm:neg_p],
  premod:yes],
Gives a paratactic analysis of adjectives in attributive (modifier) position.

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

Sentence adverbs

adv_sent_fs is_short_for
   [word_sat,
    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: <>[’L’(Token,Lexeme,p{[a1,para]}),
      ptop(Token,CTT),
      CTT qeq LTOP]]].

Only word-level verbs are allowed to be modified.

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

Subject-relative adverbs

adv_sent_subj is_short_for
   [word_sat,
Only word-level verbs are allowed to be modified.

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

Adjective-modifying adverbs (tentative treatment)

adv_adjmod is_short_for
[word_sat,
nfs content_lexitem,
token: <>Token,
lexeme: <>Lexeme,
head: adv,
spr: {},
comps: {},
head: [mod: [head: adj,
  comps: {},
  pm: neg_p],
  premod: yes],
cont: [h: [mod_hole: [key: <>AdjKey,
  ltop: <>LTOP]],
  tds: <'L'(Token, Lexeme, p([a1, para])),
  ci_a1_a1(Token, AdjKey),
  ptop(Token, CTT),
  CTT qeq LTOP]]].
allegedly

\[\text{nfs adv\_adjmod,}\]
\[\text{lexeme: <>allegedly].}\]

Complementizers

cplzer is_short_for
\[\text{[word\_sat,}\]
\[\text{slash: {},}\]
\[\text{head: [cplzer,}\]
\[\text{vform:xVform,}\]
\[\text{v\_sf:xVSF]},\]
\[\text{nfs no\_mod,}\]
\[\text{spr\_inc:xSprInc,}\]
\[\text{comps: {xSprInc}},\]
\[\text{cont: [hook: [xCH],}\]
\[\text{h: [comp\_holes: {xCH}},\]
\[\text{tds: <>[]}].\]

A complementizer is a head that only changes the head category of the complement, so to speak.

that
\[\text{nfs cplzer,}\]
\[\text{spr: {}},\]
\[\text{comps: {vform: finite,}\}
\[\text{spr: {}}]}.\]

to
\[\text{nfs cplzer,}\]
\[\text{spr: {xSpr},}\]
\[\text{comps: {vform: infinite,}\}
\[\text{spr: {xSpr}}}].\]

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_options([cont, res11, res12]).
Colour specifications

```prolog
\begin{verbatim}
    type_color(sign, '#FFFFFFE0').
    type_color(word, '#FAEBD7').
    type_color(lx, '#FA8072').
    type_color(infl_lx, '#FA8072').
    type_color(phrase, '#FFE4B5').
    type_color(head_sort, '#7FFF00').
    type_color(determin, '#FFFF33').
    type_color(nom, '#7FFF00').
    type_color(cnnoun, '#7FFF00').
    type_color(prnoun, '#7FFF00').
    type_color(perspron, '#7FFF00').
    type_color(adj, '#CCFFFF').
    type_color(adv, '#CCFFFF').
    type_color(verb, '#FFFFCC').
    type_color(v, '#FFFFCC').
    type_color(cplzer, '#FFC0CB').
    type_color(mod_sort, '#E6E6FA').
    type_color(cont_sort, '#F5FFFA').
\end{verbatim}
```

Display colours associated with the various types.

Resource locations

```prolog
\begin{verbatim}
    saved_state_file('~/home/staff/matsd/petfsg/tds1.sav').
\end{verbatim}
```

The version-dependent file name for the Prolog saved state in which the application based on this grammar will reside.

```prolog
\begin{verbatim}
    css_url('http://stp.ling.uu.se/~matsd/petfsg/3/css/pgt.css').
\end{verbatim}
```

The cascading style sheet for the output pages.

```prolog
\begin{verbatim}
    infotext(\%
        'An implementation of Token Dependency Grammar (TDS). See \', newl,
        'documentation <a href="http://stp.ling.uu.se/~matsd/tds/1">here</a>\.',
        newl, 'The PETFSG grammar tool is documented \', newl,
        '<a href="http://stp.ling.uu.se/~matsd/petfsg/3">here</a>\'].
\end{verbatim}
```

Documentation note.

End of file.
Appendix C: Auxiliary Prolog procedures

This is a listing of the additional Prolog procedures of which the application makes use. (The file name is ‘pl_tds.pl’.)

Operator declarations

\[ \text{:- op}(300, xfy, qeq). \]

Equality modulo quantifiers, Copestake et al style.

\[ \text{:- op}(200, xfy, @). \]

For token numbering.

Token labeling

\[ \text{label_token(Desc, N, Str):-} \]
\[ \hspace{1em} \text{name(N, Name),} \]
\[ \hspace{1em} \text{name(Str, StrN),} \]
\[ \hspace{1em} \text{append([116|Name], [95|StrN], NN),} \]
\[ \hspace{1em} \%\% \text{ e.g. t3_house} \]
\[ \hspace{1em} \text{name(TokId, NN),} \]
\[ \hspace{1em} \text{path_value(Desc, token, prolog_term(TokId,_)).} \]

call_prolog

call_prolog/1 provides the interface between the phrase structure rules and general Prolog computation.

\[ \text{call_prolog([mother_daughters, Mother, Dghtr1, Dghtr2]):-} \]
\[ \hspace{1em} !, \]
\[ \hspace{1em} \text{mother_daughter_relations(Mother, Dghtr1, Dghtr2).} \]

\[ \text{call_prolog([list_type_append, A, B, C]):-} \]
\[ \hspace{1em} !, \]
\[ \hspace{1em} \text{list_type_append(A, B, C).} \]

\[ \text{call_prolog([append, A, B, C]):-} \]
\[ \hspace{1em} !, \]
\[ \hspace{1em} \text{append_w_check(A, B, C).} \]

\[ \text{call_prolog(X):-} \]
\[ \hspace{1em} \%\% \text{ error handling case} \]
\[ \hspace{1em} !, \]
\[ \hspace{1em} \text{make_html(tx(’Unexpected prolog call’(X))),} \]
\[ \hspace{1em} \text{fail.} \]
mother_daughter_relations(Mtr,D1,D2):-
  path_value(Mtr,slash,MtrSlash),
  path_value(D1,slash,D1Slash),
  path_value(D2,slash,D2Slash),
  list_type_append(D1Slash,D2Slash,MtrSlash),
  empty_or_singleton(MtrSlash),
  path_value(Mtr,cont:tds,prolog_term(MtrHC,_)),
  path_value(D1,cont:tds,prolog_term(D1HC,_)),
  path_value(D2,cont:tds,prolog_term(D2HC,_)),
  append_w_check(D1HC,D2HC,MtrHC).

Computes the cont:tds value (PETFSG list) on the mother as the union of the corresponding values on the daughters. (The predicate path_value/3 is described in the PETFSG documentation.)

empty_or_singleton(list_type(list_type_e<&>fttrm)).
empty_or_singleton(list_type(list_type_ne<&>fttrm(_,list_type(list_type_e<&>fttrm)))).

**List operations**

list_type_append/3 is the append operation for PETFSG lists.

list_type_append(Arg1,Arg2,_):-
  (var(Arg1);var(Arg2)),
  !,
  grammar_error('Error (variable) in list_type_append').

list_type_append(list_type(list_type_ne<&>fttrm(Head,Tail1)),
  LIST2,
  list_type(list_type_ne<&>fttrm(Head,Tail2)):-
  !,
  list_type_append(Tail1,LIST2,Tail2).

list_type_append(list_type(list_type_e<&>fttrm),LIST2,LIST2).

list_type_append/3 is like ordinary append/3, but with an argument check as a security measure.

append_w_check(A,B,C):-
  is_list(A),
  is_list(B),
  !,
  append(A,B,C).

45
append_w_check(A, B, []):-
        print('Attempt to apply append on non-list'), nl,
        print(A), print(B), nl.

— Morphological operations

morphology(id, X, X).
morphology(affx(A), X, Y):-
        name(X, N),
        name(A, AN),
        append(N, AN, NN),
        name(Y, NN).
morphology(affx_ch_e(A), X, Y):-
        name(X, N),
        name(A, AN),
        remove_final_e(N, N2),
        append(N2, AN, NN),
        name(Y, NN).

remove_final_e(AN, AN2):-
        reverse(AN, [101|AN3]),
        reverse(AN3, AN2),
        !.

postparse_mod(cont, ALL, [SEL]):-
        !,
        path_value(ALL, cont, CONT),
        path_value(SEL, cont, CONT).

Only cont selected for output.

postparse_mod(resl1, FS, Bag2):-
        !,
        path_value(FS, cont:tds, prolog_term(TDS, _)),
        path_value(FS, cont:hook:ltop, prolog_term(Ltop, _)),
        findall(p(TDS, Tree, ImmOutsc),
                build_tree_w_globals(TDS, Tree, ImmOutsc),
Bag),
make_fs(Bag,Bag2,Ltop).

postparse_mod(resl2,FS,Bag2):-
!,
path_value(FS,cont:tds,prolog_term(TDS,_)),
path_value(FS,cont:hook:ltop,prolog_term(Ltop,_)),
findall(p(TDS,Tree,[]),
   build_tree_no Globals(TDS,Tree),
   Bag),
make_fs(Bag,Bag2,Ltop).

postparse_mod(Op,_,[]):-
!,
make_html(txbi([’No postparse op. applied.’,
                ’Option “”, Op, ”” undefined.’])).

make_fs([],[],_).

make_fs([p(TDS,Tree,ImmOutsc)|Bag],[FS|Bag2],Ltop):-
!,
collect_global_bodies(ImmOutsc, Globals),
collect_rss_info(TDS,Tree, Anchs),
path_value(FS,cont:glob,prolog_term(Globals,_)),
path_value(FS,cont:tds,prolog_term(TDS,_)),
path_value(FS,cont:tree,prolog_term(Tree,_)),
path_value(FS,cont:rss,prolog_term(Anchs,_)),
path_value(FS,cont:hook:ltop,prolog_term(Ltop,_)),
make_fs(Bag,Bag2,Ltop).

collect_rss_info(TDS,Tree, STTS3):-
collect_rss_info2(TDS,Tree, STTS),
split(STTS,STTS2),
extract_body_statements(Tree,BodyStatements),
append(BodyStatements,STTS2,STTS3).

collect_rss_info2(TDS,Tree,STTS):-
findall(paratri(Tok,TokComp,Anch),
   (subtree_paratactic(Tree,[Tok:top(_),
                             SubTree]),
    comptok(SubTree,TokComp),
    anchoring(TDS,SubTree,Anch)),
   STTS).
split([],[]).

split([paratok(Tok,TokComp,Anch)|Rest1],
     [parok(Tok,TokComp), anchor(Tok,Anch)|Rest2]):-
     split(Rest1,Rest2).

stt(Tree,Tok,CT):-
    subtree(Tree,Tok,Subtree),
    comptok(Subtree,CT).

comptok([Tok: _],Tok).

comptok([Tok: _,Tree],Tok+CT):-
    comptok(Tree,CT).

comptok([Tok: _,Tree1,Tree2],Tok+CT1+CT2):-
    comptok(Tree1,CT1),
    comptok(Tree2,CT2).

paratok([Tok1:ptop(Tok2),_],Tok1,Tok2).

paratok([_,SubTree],Tok1,Tok2):-
    paratok(SubTree,Tok1,Tok2).

paratok([_,_,SubTree],Tok1,Tok2):-
    paratok(SubTree,Tok1,Tok2).

paratok([_,_,_,SubTree],Tok1,Tok2):-
    paratok(SubTree,Tok1,Tok2).

anchoring([],_,[]).

anchoring([c1(X,Y)|TDS],Subtree,[t(Y,1,’Ind’(Y,1))|Anch]):-
    relevant_for(X,Y,Subtree),
    !,
    anchoring(TDS,Subtree,Anch).

anchoring([c2(X,Y)|TDS],Subtree,[t(Y,2,’Ind’(Y,2))|Anch]):-
    relevant_for(X,Y,Subtree),
    !,
    anchoring(TDS,Subtree,Anch).

anchoring([c3(X,Y)|TDS],Subtree,[t(Y,3,’Ind’(Y,3))|Anch]):-
relevant_for(X,Y,Subtree), !,
anchoring(TDS,Subtree,Anch).

anchoring([_|TDS],Subtree,Anch):- !,
anchoring(TDS,Subtree,Anch).

relevant_for(X,Y,Subtree):- 
\+contains(Subtree,X),
contains(Subtree,Y).

collect_global_quants([],[]).

collect_global_quants([’L’(Tok,_,ql)|Rest],[Tok|Found]):- !,
collect_global_quants(Rest,Found).

collect_global_quants([_|Rest],Found):- !,
collect_global_quants(Rest,Found).

specify_gloablity(TDRS,GLOBAL,NewBodyStatms):-
specify_gloablity2(TDRS,GLOBAL),
findall(body(X,globa1),
( member(X,GLOBAL),
  \+memberchk(body(X,globa1),TDRS)),
NewBodyStatms).

specify_gloablity2([],_).

specify_gloablity2([body(X,globa1)|TDRS],GLOBAL):-
memberchk(X,GLOBAL), !,
specify_gloablity2(TDRS,GLOBAL).

specify_gloablity2([_|TDRS],GLOBAL):- !,
specify_gloablity2(TDRS,GLOBAL).

collect_global_bodies([],[]).

collect_global_bodies([body(X,globa1)|TDRS],[X|Globals]):-
!,
collect_global_bodies(TDRS, Globals).

collect_global_bodies([_|TDRS], Globals):- !,
collect_global_bodies(TDRS, Globals).

collect_non_global_quas([], []).

collect_non_global_quas(['l'(Tok, _, q1) | Rest], ImmOutsc, [Tok:q1n(Body) | Found]):-
make_q1_node(Tok, ImmOutsc, Tok:q1n(Body)),
\+(Body=\=global), !,
collect_non_global_quas(Rest, ImmOutsc, Found).

collect_non_global_quas(['l'(Tok, _, q2) | Rest], ImmOutsc, [Tok:q2n(Restr, Body) | Found]):-
make_q2_node(Tok, ImmOutsc, Tok:q2n(Restr, Body)),
\+(Body=\=global),   % This should be true.
!,
collect_non_global_quas(Rest, ImmOutsc, Found).

collect_non_global_quas(['l'(Tok, _, LogVal) | Rest], ImmOutsc, [Tok:ptop(Para) | Found]):-
paratactic_predicate(LogVal),
makePara_node(Tok, ImmOutsc, Tok:ptop(Para)),
\+(Para=\=global),   % This should be true.
!,
collect_non_global_quas(Rest, ImmOutsc, Found).

collect_non_global_quas([_|Rest], ImmOutsc, Found):- !,
collect_non_global_quas(Rest, ImmOutsc, Found).

collect_non_global_quas([], _, []).

predicate(p(_)).

paratactic_predicate(p(Args)):-
memberechk(para, Args).

50
make_q1_node(_, [], _).

make_q1_node(Tok, [body(Tok, Body) | ImmOutsc], Tok:q1n(Body)):-
    !,
    make_q1_node(Tok, ImmOutsc, Tok:q1n(Body)).

make_q1_node(Tok, [], ImmOutsc, Tok:q1n(Body)):-
    !,
    make_q1_node(Tok, ImmOutsc, Tok:q1n(Body)).

make_q2_node(_, [], _).

make_q2_node(Tok, [body(Tok, Body) | ImmOutsc],
              Tok:q2n(Restr, Body)):-
    !,
    make_q2_node(Tok, ImmOutsc, Tok:q2n(Restr, Body)).

make_q2_node(Tok, [restr(Tok, Restr) | ImmOutsc],
              Tok:q2n(Restr, Body)):-
    !,
    make_q2_node(Tok, ImmOutsc, Tok:q2n(Restr, Body)).

make_q2_node(Tok, [], ImmOutsc, Tok:q2n(Restr, Body)):-
    !,
    make_q2_node(Tok, ImmOutsc, Tok:q2n(Restr, Body)).

make_para_node(Tok, [ptop(Tok, Para) | _],
               Tok:ptop(Para)):-
    !.

make_para_node(Tok, [], ImmOutsc, Tok:ptop(Para)):-
    !,
    make_para_node(Tok, ImmOutsc, Tok:ptop(Para)).

collect_leaves([], []).

collect_leaves(['L'(Tok, _, LogVal) | Rest],
               [{Tok:[]} | Found]):-
    predicate(LogVal),
    \+paratactic_predicate(LogVal),
    !,
    collect_leaves(Rest, Found).

collect_leaves([_|Rest], Found):-
!,
collect_leaves(Rest,Found).

tdr_kinds([],[],[],[],[],[]).

tdr_kinds([’L’(A,B,C)|Rest],[’L’(A,B,C)|LOS],
    QEQS,ImmOutsc,CIS):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([A eq B|Rest],LOS,
    [A eq B|QEQS],ImmOutsc,CIS):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([ptop(A,B)|Rest],LOS,
    QEQS,[ptop(A,B)|ImmOutsc],CIS):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([body(A,B)|Rest],LOS,
    QEQS,[body(A,B)|ImmOutsc],CIS):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([restr(A,B)|Rest],LOS,
    QEQS,[restr(A,B)|ImmOutsc],CIS):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([c1(A,B)|Rest],LOS,
    QEQS,ImmOutsc,[os(A,B)|CIS]):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([c2(A,B)|Rest],LOS,
    QEQS,ImmOutsc,[os(A,B)|CIS]):-
    !,
    tdr_kinds(Rest,LOS,QEQS,ImmOutsc,CIS).

tdr_kinds([c3(A,B)|Rest],LOS,
    QEQS,ImmOutsc,[os(A,B)|CIS]):-
    !,
tdr_kinds(Rest, LOS, QEQS, ImmOutsc, CIS).

build_tree_w Globals(TDRS, Tree, ImmOutsc2):-
  tdr_kinds(TDRS, LOS, QEQS, ImmOutsc, CIS),
  collect_global_quants(LOS, GLOBAL),
  specify_globality(TDRS, GLOBAL, NewBodyStatms),
  append(ImmOutsc, NewBodyStatms, ImmOutsc2),
  collect_leaves(LOS, LEAVES),
  build_tree(_, Tree, QUAS, LEAVES, [], []),
  check_tree(Tree, GLOBAL, QEQS, CIS).

build_tree_noGlobals(TDRS, Tree):-
  tdr_kinds(TDRS, LOS, QEQS, ImmOutsc, CIS),
  collect_non_global_quas(LOS, ImmOutsc2, QUAS),
  collect_leaves(LOS, LEAVES),
  build_tree(_, Tree, QUAS, LEAVES, [], []),
  check_tree(Tree, [], QEQS, CIS).

build_tree(T, [T:[]], QUAS, LEAVES, QUAS, LEAVES2):-
  select_any(T:[]), LEAVES, LEAVES2).

build_tree(T, [T:q1n(Body), Ptree], QUAS, LEAVES, QUAS3, LEAVES2):-
  select_any(T:q1n(Body), QUAS, QUAS2),
  build_tree(Body, Ptree, QUAS2, LEAVES, QUAS3, LEAVES2).

build_tree(T, [T:q2n(Restr, Body), Rtree, Btree], QUAS, LEAVES, QUAS4, LEAVES3):-
  select_any(T:q2n(Restr, Body), QUAS, QUAS2),
  build_tree(Restr, Rtree, QUAS2, LEAVES, QUAS3, LEAVES2),
  build_tree(Body, Btree, QUAS3, LEAVES2, QUAS4, LEAVES3).

build_tree(T, [T:ptop(Para), Ptree], QUAS, LEAVES, QUAS3, LEAVES2):-
  select_any(T:ptop(Para), QUAS, QUAS2),
  build_tree(Para, Ptree, QUAS2, LEAVES, QUAS3, LEAVES2).


53
select_any(EP, [X|REST1], [X|REST2]):-
    select_any(EP, REST1, REST2).

check_tree(Tree, GLOBAL, QEQS, CIS):-
    !,
    check_tree_qeqs(QEQS, Tree),
    check_tree_cis(CIS, Tree, GLOBAL).

check_tree(Tree, GLOBAL, QEQS, CIS, R1, R2):-
    !,
    check_tree_qeqs(QEQS, Tree, R1),
    check_tree_cis(CIS, Tree, GLOBAL, R2).

check_tree_qeqs([], _).

check_tree_qeqs([T1 qeq T2|QEQS], Tree):-
    !,
    qeq(T1, T2, Tree),
    check_tree_qeqs(QEQS, Tree).

check_tree_qeqs([], _, ok).

check_tree_qeqs([T1 qeq T2|QEQS], Tree, Res):-
    qeq(T1, T2, Tree),
    !,
    check_tree_qeqs(QEQS, Tree, Res).

check_tree_qeqs([T1 qeq T2|_], _, failing(T1 qeq T2)).

qeq(T, T, _):-
    !.

qeq(T1, T2, [T1:q2n(_,Tx),_,Body]):-
    qeq(Tx, T2, Body).

qeq(T1, T2, [T1:q1n(Tx), Body]):-
    qeq(Tx, T2, Body).

qeq(T1, T2, [_, Subtree]):-
    qeq(T1, T2, Subtree).

qeq(T1, T2, [_, Subtree, _]):-
qeq(T1,T2,Subtree).
qeq(T1,T2,[_,_,Subtree]):-
qeq(T1,T2,Subtree).
check_tree_cis([],_,_).
check_tree_cis([os(T1,T2)|CIS],Tree,GLOBAL):-
outscopes(os(T1,T2),Tree,GLOBAL),
check_tree_cis(CIS,Tree,GLOBAL).
check_tree_cis([],_,_,ok).
check_tree_cis([os(T1,T2)|CIS],Tree,GLOBAL,Res):-
outscopes(os(T1,T2),Tree,GLOBAL),!
check_tree_cis(CIS,Tree,GLOBAL,Res).
check_tree_cis([os(T1,T2)|_],_,_,failing(os(T1,T2))):-!
outscopes(os(T1,_),_,GLOBAL):-
memberchk(T1,GLOBAL).
outscopes(os(T1,T2),Tree,_):-
outscopes_in_tree(T1,T2,Tree).
outscopes_in_tree(T1,T2,Tree):-
subtree(Tree,T1,Subtree),
contains(Subtree,T2).
subtree([T1:X|Rest],T1,[T1:X|Rest]).
subtree([_,Restr, _],T1,Subtree):-
subtree(Restr,T1,Subtree).
subtree([_,_,Body],T1,Subtree):-
subtree(Body,T1,Subtree).
subtree([_,Para],T1,Subtree):-
subtree(Para,T1,Subtree).
subtree_paratactic([T1:ptop(X)|Rest],[T1:ptop(X)|Rest]).
subtree_paratactic([_,Restr,_*],Subtree):-
    subtree_paratactic(Restr,Subtree).

subtree_paratactic([_,_,Body],Subtree):-
    subtree_paratactic(Body,Subtree).

subtree_paratactic([_,Para],Subtree):-
    subtree_paratactic(Para,Subtree).

contains([T:__],T).

contains([_,Subtree],T):-
    contains(Subtree,T).

contains([_,Subtree,_],T):-
    contains(Subtree,T).

contains([_,_,Subtree],T):-
    contains(Subtree,T).

extract_body_statements([Q:q2n(_,Body),T1,T2],
    [body(Q,Body)|More]):-
    !,
    extract_body_statements(T1,B1),
    extract_body_statements(T2,B2),
    append(B1,B2,More).

extract_body_statements([Q:q1n(Body),T1],
    [body(Q,Body)|More]):-
    !,
    extract_body_statements(T1,More).

extract_body_statements([_,T1],More):-
    !,
    extract_body_statements(T1,More).

extract_body_statements([_],[]):-
    !.

Prolog term displaying

disp_prolog_term(T,ProVarNrI,ProVarNrO):-
    instantiate_vars(T,ProVarNrI,ProVarNrO),

print('<tt>'),
print_prolog_terms(T),
print('</tt>').

instatiate_vars(T,ProVarNrI,ProVarNrO):-
    var(T),
    !,
    make_symbol_for_var(T,ProVarNrI,ProVarNrO).

instatiate_vars(T,ProVarNr,ProVar):-
    simple(T),
    !.

instatiate_vars([H|T],ProVarNrI,ProVarNrO):-
    !,
    instantiate_vars(H,ProVarNrI,ProVarNm),
    instantiate_vars(T,ProVarNm,ProVarNrO).

instatiate_vars(T,ProVarNrI,ProVarNrO):-
    !,
    T =.. List,
    instantiate_vars(List,ProVarNrI,ProVarNrO).

make_symbol_for_var(T,ProVarNrI,ProVarNrO):-
    name(ProVarNrI,Name),
    append([60,117,62,80|Name],
           [60,47,117,62],TT),
    name(T,TT),
    ProVarNrO is ProVarNrI + 1.

print_prolog_terms(X):-
    var(X),
    !,
    print(X).

print_prolog_terms([X:Y|Z]):-
    !,
    print_tree([X:Y|Z],0).

This case finds contain.

print_prolog_terms(X):-
    is_list(X),
    !,
print_list(X).

print_prolog_terms(X):-
  print(X).

print_list([X|Rest]):-
  Rest\==[],
  \+atom(X),
  !,
  print('<pre>'), nl,
  print([''), print(X), print(', '), nl,
  print_list_cont(Rest).

print_list(X):-
  !,
  print(X).

print_list(X):-
  !,
  print(X).

print_list_cont([X]):-
  !,
  print(' '), print(X), print(']'), nl,
  print('<pre>').

print_list_cont([X|Rest]):-
  !,
  print(' '), print(X), print(', '), nl,
  print_list_cont(Rest).

print_tree(X,Level):-
  print('<pre>'), nl,
  print_tree2(X,Level), nl,
  print('</pre>').

print_tree2([A],Level):-
  !,
  indent(Level), print('['), print(A), print(']').'.

print_tree2([A,B],Level):-
  !,
  indent(Level), print('['), print(A), print(',', nl,
Level2 is Level + 1,
print(' '), print_tree2(B,Level2), print(']')

print_tree2([A,B,C],Level):-
  !,
  indent(Level), print('['), print(A), print('),') , nl,
  Level2 is Level + 1,
  print(' '), print_tree2(B,Level2), print('),') , nl,
  print(' '), print_tree2(C,Level2), print(']')

print_tree2(A,Level):-
  !,
  indent(Level), print('*'), print(A).

indent(0):-
  !.

indent(X):-
  print(' '),
  XX is X - 1,
  indent(XX).

End of file.

Appendix D: Downloadable TDS grammar files

These files are described above in this document. They are found in http://stp.ling.uu.se/~matsd/tds/1/f/.

gram_tds.pl: The TDS grammar.

lex_tds.pl: More lexical entries.

pl_tds.pl: Contains SICStus Prolog procedures used by the TDS grammar.
suites_tds.pl: Defines test suites.

make_tds.pl: Used to create the TDS grammar saved state.