Reports from
Uppsala University
Department of Linguistics

RUUL
nr 34

Three Papers on Computational Syntax and Semantics

Mats Dahllöf

UPPSALA UNIVERSITY

September
1999

Reports from
Uppsala University
Department of Linguistics

RUUL
nr 34

Three Papers on Computational Syntax and Semantics

Mats Dahllöf

Prolog-embedding typed feature structure grammar

Implementing token-based minimal recursion semantics

Predicate-functor logic

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

September 1999
ISBN: 91-973737-0-2
ISSN: 0280-1337

RUUL (Reports from Uppsala University, Department of Linguistics), nr 34, September 1999.

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

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

Typeset in LaTeX by the author.

ISBN: 91-973737-0-2 (this volume).
ISSN: 0280-1337 (*RUUL* series).

Printed by *Repro Ekonomikum*, Uppsala.

Table of contents

Preface

This number of *RUUL*, Reports from the Uppsala University Department of Linguistics, contains three papers on computational syntax and semantics by Mats Dahllöf. The first one documents a feature-structure grammar system—a formalism and a parser—called 'Prolog-embedding typed feature structure grammar'. The system is put to use in the second paper, 'Implementing token-based minimal recursion semantics'. A companion piece, 'Token-based minimal recursion semantics', which is intended for publication elsewhere, presents the theoretical background and motivation. (At present, it is available from the author.) The implementation report in this volume gives the technical details. The third paper, 'Predicate-functor logic', is an introduction to Quine's *Predicate-functor logic* and its interesting variable-free treatment of quantification. (Some of this material was also used in my dissertation *On the Semantics of Propositional Attitude Reports*, Dept of Linguistics, Göteborg University, 1994.)

Uppsala, September 1999

Mats Dahllöf

RUUL (Reports from Uppsala University Dept of Linguistics) **34**, 3–37.

Prolog-embedding typed feature structure grammar

Mats Dahllöf
Department of Linguistics, Uppsala University
E-mail: `Mats.Dahllof@ling.uu.se`
September, 1999

1 Introduction

This paper describes a Prolog-based typed feature structure grammar formalism and a chart parser for it. The system provides some facilities for tracing the application of the grammar, including a graphical interface to the chart. It is intended to be tool for the implementation of constraint-based grammars, useful for both research and teaching purposes.

The formalism will be referred to as "Prolog-embedding typed feature structure grammar" (PETFSG). The formalism is a typed and enriched variant of PATR-II (Shieber 1986). It uses typed feature structures as its data type, but these allow the embedding of ordinary (untyped) Prolog terms. Prolog terms may consequently be used whenever convenient, for instance, to make semantic representations more compact, and in cases where the inventory of values is very large or hard to predict (semantics). Typing may at the same time be used to impose restrictions wherever useful. The PETFSG formalism allows Prolog calls to be made from inside the grammar. In this way, Prolog computation is allowed to take care of restrictions

which can not be expressed by means of simple feature unification. This combination of typed FSs and Prolog terms and of unification and general computation makes the formalism powerful, flexible, simple, and easy to use. PETFSG is defined in SICStus Prolog.

The PETFSG formalism and system have been designed with Head-Driven Phrase Structure Grammar (HPSG, Pollard and Sag 1994) in mind. The PETFSG system may be compared to ALE (Carpenter 1997). The type system of PETFSG is more restrictive than some of those used in the HPSG literature.

2 The structure of a PETFSG grammar

A PETFSG grammar contains:

- precisely one *type hierarchy declaration*
- at most one *feature declaration* for each type
- any number of FS *definitions*
- any number of *grammar rules*
- any number of *word form entries*
- any number of *inflectional patterns*
- any number of *lexeme entries* (each defining a number of word forms with the help of an inflectional pattern)
- any number of *lexical rules*
- any number of *internal predicate definitions*

The items of the grammar are all Prolog terms, each delimited by a full stop. The grammar file (or set of files) is compiled to an

internal form by the parsing system. During the compilation the internal coherence of the grammar is checked, and error messages are issued if there is something wrong with type or feature declarations and whenever grammar rules and lexical entries fail to agree with them. Faulty items in the grammar are otherwise ignored by the system.

2.1 Declaring the type hierarchy

A PETFSG type system is a simple type hierarchy without cross-classification (i.e. a tree structure). Every type is consequently the immediate subtype of at most one type. The type system is based on a set of named most general types. The most general type, which subsumes all types, is not named unless it is explicitly declared (in which case the set of most general named types is singleton). Features may be associated with any named type.

The type system is declared by a list of terms with '`subsumes`' as the main (infix) operator. These `subsumes`-terms are of the form '*Type* `subsumes` *Subtypes*' where *Type* is an atom which names a type and *Subtypes* is a list of `subsumes`-terms. This term defines the type *Type*. The terms in *Subtypes* defines the set of immediate subtypes to *Type*. The `subsumes`-terms of *Subtypes* are interpreted in the same way. This notation allows us to represent the entire type hierarchy as a list of `subsumes`-terms. The most specific types, i.e. those without subtypes, are consequently defined by terms of the form '*Type* `subsumes []`'.

A term of the form '`type_hierarchy(`*TH*`)`' declares *TH* as the type hierarchy. (*TH* is a list of `subsumes`-terms, as described above.) There is precisely one such term in a PETFSG grammar, and it must be the first term.

A very simple Sample Grammar (listed in Appendix 1) will be used here for illustration. This is its type hierarchy declaration. (Readability is improved by a suitable indentation.)

```
type_hierarchy([
   sign subsumes [],
   head_type subsumes [
      numbrd subsumes [
         n subsumes [],
         det subsumes []],
      v subsumes []],
   tense_type subsumes [
      present subsumes [],
      past subsumes []],
   number_type subsumes [
      sing subsumes [],
      plur subsumes []]]).
```

Apart from the user-defined types provided by the '`type_hierarchy`' declaration, PETFSG recognizes *list* and *Prolog term* as separate types.

The data structures of the grammar will, as previously mentioned, be called *feature structures* (henceforth FSs), and are, as we have seen, of three main kinds: *attribute-value* FS*s*, which are values of the user-defined types, *lists*, and *Prolog terms*.

In PETFSG, lists are represented as Prolog lists prefixed by '`$`'. The Prolog terms are of the form '`<>_`'. (The difference matters in the formulation of grammar rules, where lists are treated in a special way. See below. Prolog lists prefixed by '`<>`' are treated like "ordinary" Prolog terms by PETFSG.)

2.2 Representing types and values

The PETFSG-formalism represents 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`'. Specific Prolog term values are of the form '`<>` *X*'.
- The name of the type of lists is '`list`'. Specific list values are of the form '`$` *X*'.

2.3 Declaring features

Features are associated with (user-defined) types by means of declarations of the form '`features(`*Type*`,`*Features*`)`', where *Type* is a type and *Features* is a list of declarations of the features that are associated with the type (and all its subtypes). Every feature name is in this way associated with precisely one type and is thereby declared as appropriate for this type and all its subtypes. A feature declaration 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.

If there is no feature declaration for a type, *Type*, no feature is associated with it. This situation can be made explicit by a declaration of the form '`features(`*Type*`,[])`'.

The Sample Grammar contains the following feature declarations.

```
features(sign, [head:head_type,
                spr:list,
                sem:prolog_term]).
```

```
features(numbrd, [number:number_type]).

features(v, [tense:tense_type]).
```

2.4 Specifying values

The type and feature values of an attribue-value FS are specified by means of the relations `===/2` and `denotes/2`. The two relations "mean" the same, but are used differently. Calls to '`===`' are presupposed to succeed. Otherwise, a warning is issued. The relation `denotes/2` (infix) is used to test values in contexts where failures are anticipated. It is suitable to use the predicate `===/2` (also infix) to specify constraints in relation to lexical entries and grammar rules (which, of course, are intended to be internally coherent).

The internal structure of the FSs is not intended to be "visible" in the grammar. The predicates `===/2` and `denotes/2` allow us to assign values to features and to "hide" their internal structure. (This *kind* of Prolog reconstruction of PATR-II-style notation is described by Gazdar and Mellish 1989.) The arguments to the two relations are translated into the internal representation of the PETFSG-parser. The following kinds of argument are allowed and they are understood as follows, `===/2` and `denotes/2` being sensitive to the *current binding* of their arguments:

- *Variables*: A variable stands for any kind of object. Cooccurrence of variables indicate sharing of substructure in the usual Prolog way.
- *Type names*: A type name stands for a FS which is of the type in question and not further specified.

- *Path expressions*: A path expression is a term of the form '*PathExp* : *Feat*', where *Feat* is a feature name and *PathExp* is a path expression or a FS (represented by a variable in the "normal" case). A path expression denotes the value of the given path within the given FS in the usual fashion (cf. Sheiber 1986). For instance, '`A:f:g:h`' would be the value of the feature '`h`' in the value of the feature '`g`' in the value of the feature '`f`' in the FS represented by the variable '`A`'.

- `<>`-*terms*: A `<>`-term (of the form `<>`X, where X is any Prolog term) is treated like an ordinary Prolog term.

- `$`-*terms*: A `$`-term (of the form `$`L, where L is any Prolog term subsuming a list) stands for a list of arbitrary FSs. The list structure can be indicated by means of the Prolog symbol '`|`'. The elements of the list can be characterized by further `===/2` or `denotes/2` statements.

- A term of the form `nfs(`X`)` (Named Feature Structure) stands for the FS that has been given the name X (by an `is_short_for` item). See below, section 2.5.

- The symbol '§' is used in word form entries, lexeme entries, inflectional pattern definitions, and feature structure definitions to stand for the FS associated with the word form description about to be defined or the FS about to be named.

Note that sharing of substructure is indicated in the standard Prolog way by means of variables.

2.5 FS definitions

A FS may be named and thereafter recalled. This is useful in cases where the same complex structure is used several times in the grammar. A structure is given a name by means of a grammar item of the form '*Name* `is_short_for` *Conds*'. In the *Conds* context, '§' stands for the FS to be named. ('§' is consequently a pronoun-like device.) (Note that the SICStus Prolog syntax requires a space between '§' and the ':' operator.)

This is an example (from the Sample Grammar):

```
present_plural is_short_for
      § :head === v,
      § :head:tense === present,
      § :spr === $[Subj],
      Subj:head:number === plur.
```

2.6 Grammar rules

The PETFSG rules are a kind of enriched phrase structure rules, allowing also lists and Prolog calls to occur in the right-hand sequence. The left-hand item of a rule is a FS (corresponding to the mother node). The right-hand term is a list, each of whose members is of one of the following three kinds:

- Terms of the form '`!`F', where F is a FS, match a single ordinary syntactic daughter (the "standard" kind of right-hand item, in other words.)
- Terms of the form '`+`L', where L is a PETFSG list, match a number of syntactic daughters, one for each element of L and in the same order. (The formulation '`+ $[A]`' is consequently equivalent to '`!A`'.) When a '`+`L' term occurs in this context, L is presupposed to be a list.

- Terms of the form '`?`T', when encountered by the parser, causes the parser to call the Prolog term T. Every solution to the `?`-call (or sequence of `?`-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.)

Grammar rules are of the form 'LH `===>` RH `where` *Conds*', which connects the left-hand item, LH, with the right-hand list, RH, of a rule. Constraints on the FSs involved are formulated as a Prolog condition, *Conds*. In this context, the '`===`' predicate is useful, but other kinds of condition may occur. For instance, collections of '`===`'-constraints may be defined as predicates directly in Prolog.

The conditions of the '`===>`'-rules are evaluated during compilation. Each instance of 'LH`===>`RH' resulting from the evaluation of *Conds* will be stored as a compiled grammar rule in the database.

Returning to our Sample Grammar, we find the following rule:

```
Mtr ===> [!Spr,
          !Head,
          ?compute_semantics(Spr,Head,Mtr)] where
        Head:spr === $[Spr],
        Mtr:head === Head:head.
```

(This rule assembles a specifier and a head. When the two constituents have been found by the parser, the Prolog call '`compute_semantics(Spr,Head,Mtr)`' is made.) The compiled form of this rule is the fact 'LH `===>` RH', that is defined when the *Conds* part of the rule is evaluated.

2.7 Word form entries

A word form entry is of the form '*Form* `>>>` *Conds*'. *Form* is the term representing the shape of the word. The system assumes that words are represented as Prolog atoms, capturing their orthographic forms. The condition, *Conds*, of an entry is evaluated during compilation. In this context, '§' stands for the FS describing the lexical item.

The following lexical entries (from the Sample Grammar) provide some illustration:

```
every >>>
        § :head === det,
        § :head:number === sing,
        § :spr === $[],
        § :sem === <> every.

sleep >>>
        § === nfs(present_plural),
        § :spr === $[Subj],
        § :sem === <> sleep,
        Subj:head === n,
        Subj:spr === $[].
```

2.8 Inflectional patterns

The PETFSG formalism and system supports morphological analysis of a primitive (but in principle powerful) kind.

Inflectional patterns are of the form '*Name* `is_pattern` *FList*', where *Name* is the name of the pattern and *FList* a list defining the forms generated by the pattern. The elements of *FormList* are terms of the form '`p(`*Morph*`,`*Conds*`)`' (one for each word form defined). Here,

Morph is a morphological operation and *Conds* a condition specifying the FS associated with the word form in question. ('§' stands for this FS.)

The following pattern (from the Sample Grammar) illustrates the idea. (The morphology is described below.) It generates a singular form, identical to the stem, and a plural form, with an "s" affixed:

```
noun_pattern is_pattern
   [p(id,
      (§ :head === n,
       § :head:number === sing)),
    p(affx(s),
      (§ :head === n,
       § :head:number === plur))].
```

2.8.1 Defining morphological processes

Morphological processes are defined *directly in Prolog*, by means of the predicate `morphology/3`, relating a morphological process, an input form, and an output form. The morphological process is characterized by an arbitrary Prolog term.

For instance, the following clauses defines the morphology needed above ('`id`' is identity, '`affx(`*A*`)`' adds affix *A*, atoms being used to represent words):

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

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

2.9 Lexeme entries

In a lexeme entry, a stem is associated with an inflectional pattern and a description applying to each of its forms. Such an entry is of the form '*Stem* `of_pattern` *Name* `>>>` *Conds*', where *Stem* is the input to the morphological processes of the (previously defined) inflectional pattern by the name *Name*. *Conds* is a condition specifying the FSs associated with each of the generated word forms. When the PETFSG compiler encounters a lexeme entry, the resulting word forms are generated and stored in the lexical database.

This is an example of a lexeme entry:

```
dog of_pattern noun_pattern >>>
        § :head === n,
        § :head:number === Num,
        § :spr === $[Spr],
        § :sem === <> dog,
        Spr:head === det,
        Spr:head:number === Num.
```

When encountered by the compiler, this item causes two word forms to be stored in the lexicon.

2.10 Lexical rules

New lexical entries may be derived by means of HPSG-style lexical rules. A PETFSG lexical rule involves two feature structures and a morphological process. Whenever the first unifies with the description of a lexical entry, the rule "generates" a new entry from the second term 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. The shape of the new entry

is given by the morphological process, which is applied on the shape of the input entry. As is the case with grammar rules and (explicit) lexical entries, constraints on the feature structures involved are formulated as Prolog conditions.

A lexical rule may, for instance, look like the one in the Sample Grammar:

```
lexical_rule(A,B,affx(s)) where
        A:head === v,
        A:head:tense === present,
        A:spr === $[ASubj],
        B:spr === $[BSubj],
        ASubj:head:number === plur,
        BSubj:head:number === sing,
        A:head === B:head,
        A:sem === B:sem,
        BSubj:spr === ASubj:spr.
```

This rule derives a singular agreement present tense verb form from the corresponding plural form (third person being presupposed in the toy Sample Grammar).

The application of the lexical rules is triggered by means of a '`:-expand_lexicon(`N`)`' command, where N, is the number of times the lexical rules are allowed to be applied (see Section 3.2).

2.11 Internal predicate definitions

Some Prolog predicates of the PETFSG system may be redefined in the grammar. In this way, the user may modify some procedural features of the system.

There are three predicates which may be redefined in this way: `report_look_up/2`, `token_description/3`,

and `fs_output_form/3`. A term of the form '`redefine_predicate(`*Clause*`)`' redefines one of these predicates.

The predicate `report_look_up/2` is called when the lexical descriptions of an input string item have been assembled. Its two arguments is the item in question and the number of lexical hits. Its default definition is '`report_look_up(_,_)`', i.e. it succeeds without making any difference. When a redefinition of it is encountered by the PETFSG compiler, the previous (default or user-defined) definition of it is retracted, and the new one asserted. It is presupposed that precisely one clause defines this predicate. By redefining `report_look_up/2`, the user may have the system report lexical look-up information when parsing is performed, e.g. as follows:

```
redefine_predicate(
  (report_look_up(L,N):-
     (print(L),
      print(': '),
      print(N),
      print(' hit(s).'),
      nl))).
```

The predicate `fs_output_form/3` selects or modifies the information to be printed when the predicate `pp/2` (parse and print, see section 3.4) is used. '`fs_output_form(`*SelectionType*`,`*FS1*`,`*FS2*`)`' holds when *FS2* is related to *FS1* in the way named *SelectionType*. If *FS1* is the FS representing the actual description, then *FS2* is the FS that is printed. This allows, for instance, a subset of the feature values to be selected for printing. The default definition is '`fs_output_form(all,X,X)`', i.e. identity is named '`all`'. (The FSs are printed without being modified.) See Dahllöf (1999,

this volume) for further illustration. It is presupposed that there is one clause in the definition of `fs_output_form/3` per selection type name. When a redefinition of it is encountered by the PETFSG compiler, the system retracts the clause for the given selection type name, if there is one already.

The predicate `token_description/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 graphical form (an atom). The default definition of `token_description/3` is '`token_description(_,_,_)`', i.e. it succeeds without making any difference. Dahllöf (1999, this volume) makes use of a more substantial version of the predicate. When a redefinition of it is encountered by the PETFSG compiler, the previous (default or user-defined) definition of it is retracted, and the new one asserted. It is presupposed that precisely one clause defines this predicate.

Attempts to use '`redefine_predicate(`*Clause*`)`' terms to redefine other predicates result in an error.

3 The PETFSG parser

3.1 Compiling a grammar

The PETFSG grammar must be converted to the internal form used by the parser by means of the predicate `compile_grammar/1`. In the process, the grammar is also checked for consistency. (An overview of compilation error messages is given in Appendix 2.)

The compilation must be preceded by a '`:-prepare_compilation`' command. The predicate `prepare_compilation/0` erases the grammar, if there is

one already loaded into the system, and prepares the system for the compilation.

The grammar entries should (for obvious reasons) be compiled in the following order: (1) The type hierarchy, (2) the feature declarations, and (3) the other grammar items. Named FSs and inflectional patterns must be defined before they are referred to. Error messages and warnings are issued when contradictions and errors occur (cf. Appendix 2). User-defined Prolog predicates used to define grammar rules and lexical entries must, of course, be defined before they are used. The contents of a grammar file are compiled by the call '`compile_grammar(`*File*`)`', where *File* is the atom naming the file to be compiled. The grammar may be dispersed over any number of files.

3.2 Expanding the lexicon

The lexical rules are treated as redundancy rules, in the sense that the lexicon is expanded before the parser is put to work. This process is triggered by a '`:-expand_lexicon(`N`)`' command. The integer N defines the number of times the lexical rules are applied. The command '`:-expand_lexicon(1)`' means that every lexical rule is matched against every (explicit) lexical entry, and '`:-expand_lexicon(2)`' means that there will also be 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 (as defined by the sharing in the lexical rule) is added to the lexicon. The new entry is not compared to the already present ones before being added.

3.3 Parsing mechanism

The PETFSG parser is an ordinary chart parser (cf. Gazdar and Mellish 1989: Ch 6). This mechanism allows us to apply the grammar to strings (needless to say) and to inspect how it works or fails to work. The locations of words and substrings are defined by pairs of vertices. A vertex is an integer: 0 is the beginning of the input string. 1 separates the first word from the second, etc. Inactive edges are defined by terms of the form '`inactive_edge(`*A,B,Desc*`)`', where *A* is the start vertex, *B* is the end vertex, and *Desc* is the associated feature structure (linguistic description).

Active edges are of the form '`active_edge(`*A,B,Desc,Acts*`)`'. The fourth item *Acts* is an action list, defining what is required to complete the assemblage of an expression. Actions are represented by terms of three kinds: `!`-terms, `+`-terms, and `?`-terms. `!`-terms represent required daughter constituents. `+`-terms represent a sequence of required daughter constituents. `?`-terms represent Prolog calls.

A *task* comes into being whenever an active and inactive edge meet (i.e. when `active_edge(_,V,_,_)` and `inactive_edge(V,_,_)`). Tasks are the basic units of processing in the parser.

A grammar rule *LH* `===>` *RH* is said to be *invoked* at a vertex, *V*, whenever an `active_edge(`*V,V,LH,RH*`)` is added to the chart. (This means that the parser initiates the assemblage of a constituent built according to the rule in question.)

The parsing process proceeds as follows:

- The input string is a list of Prolog atoms. The first step of the parser is to perform lexical analysis. It adds one inactive edge (with the appropriate span) for each lexical description of each item in the input list.

- The second step is to invoke *each* grammar rule at the first vertex (0) of the chart.
- Each task (i.e. pair of an active and inactive edge that meet) are then evaluated. When a new active edge is added, all relevant rules are invoked. (When `active_edge(_,`*V*`,_,[!` *ToFind*`|_])` or `active_edge(_,`*V*`,_,[+ $[`*ToFind*`|]|_])` is added, each rule *LH* `===>` *RH* such that *LH* is compatible with *ToFind* is invoked at *V*.)

The parsing process stops when all tasks have been evaluated. The grammar may however 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 does not normally occur in grammars considered adequate for a natural language.

3.4 Parsing and chart inspection predicates

The predicate `pp/2` takes two arguments, a selection type name, whose significance is as specified by the `fs_output_form/3` (see section 2.11) and an input list. A call '`pp(`*SelectionType*`,`*Str*`)`' causes *Str* to be parsed, and the resulting feature structures to be printed as required by the *SelectionType* case of `fs_output_form/3`. If the `fs_output_form/3` call fails, `pp/2` prints '`fs_output_form failure on this one`'.

For instance:

```
| ?- pp(all,[every, dog, sleeps]).
----- [every,dog,sleeps]
----- 1 parses. all selection:
```

```
* sign #1
  HEAD: * v #2
    TENSE: * present #3
  SEM: <> ((every@dog)@sleep) #4

yes
| ?-
```

This and subsequent examples presuppose that the the Sample Grammar is loaded into the PETFSG system.

In the output form each feature structure is introduced by a '$*$' and a specification of its type. A number, preceeded by '#' is used to refer to each substructure, and is of use in case the substructure reenters the main structure by being shared. The anaphoric references are of the form '^*Number*' (examples are found below). Feature names are converted to capitals to enhance readability. (They are, otherwise, Prolog atoms.)

A survey of the edges forming the chart is given by the predicate `edge_count/0`. Its output looks like this:

```
| ?- edge_count.
0 to 0: 0 inactive, 1 active over [].
0 to 1: 1 inactive, 1 active over [every].
0 to 2: 1 inactive, 1 active over [every,dog].
0 to 3: 1 inactive, 1 active over [every,dog,sleeps].
1 to 1: 0 inactive, 1 active over [].
1 to 2: 1 inactive, 1 active over [dog].
2 to 2: 0 inactive, 1 active over [].
2 to 3: 1 inactive, 1 active over [sleeps].
3 to 3: 0 inactive, 1 active over [].

yes
| ?-
```

The content of edges may be printed by means of the predicates `pri/2` (print inactive edge) and `pra/2` (print active edge), which take two node numbers as arguments and print the edges having the span in question. For instance:

```
| ?- pri(0,2).
--- Inactive edge(s) from 0 to 2:
  * sign #1
  HEAD: * n #2
    NUMBER: * sing #3
  SEM: <> (every@dog) #4

yes
| ?- pra(1,2).
--- Active edge(s) from 1 to 2:
   DESC: * sign #1
         HEAD: * head_type #2
   ACTIONS:
   ! * sign #3
     HEAD:  ^2
     SPR: ----- #4 ---<
         **** sign #5
            HEAD: * n #6
              NUMBER: * sing #7
            SPR: ----- #8 ---<
                **** sign #9
                   HEAD: * det #10
                     NUMBER:  ^7
              >--------
            SEM: <>dog #11
        >--------
   ?-call compute_semantics(...)
```

```
yes
| ?-
```

Lists are printed with their items arranged vertically, between '`----- #`*N* `---<`' and '`>--------`', *N* being the number indexing the entire list token.

The parser stores the tasks that have been evaluated. This makes it possible to trace what has happened during the parsing process. The predicate `prtasks/4` is used to access this information. The active edge has an action list, whose head is a term of the form '`!`*Desc*' or (as it were) '`$[`*Desc*`|_]`' (i.e. *Desc* corresponds to the constituent that is to be found. The task fails if *Desc* is incompatible with the description on the inactive edge. Those failed tasks whose active and inactive edges span *A* to *B* and *B* to *C* (respectively) are listed by a call '`prtasks(`*A,B,C*`,fail)`'. An explanation of the unification failure is also given.

```
| ?- pp(all,[every,dogs]).
----- [every,dogs]
----- 0 parses. all selection:
yes
| ?- prtasks(0,1,2,fail).
--- TASK  concerning [every] + [dogs]
========== TRYING TO ASSEMBLE:
      * sign #1
     HEAD: * head_type #2
========== REQUIRED ITEM IS:
      * sign #3
     HEAD:  ^2
     SPR: ----- #4 ---<
```

```
         **** sign #5
            HEAD: * det #6
              NUMBER: * sing #7
            SPR: []
            SEM: <>every #8
       >--------
========== INACTIVE EDGE FOUND, FROM 1 TO 2:
      * sign #9
     HEAD: * n #10
       NUMBER: * plur #11
     SPR: ----- #12 ---<
         **** sign #13
            HEAD: * det #14
              NUMBER:  ^11
       >--------
     SEM: <>dog #15
F A I L U R E: sing contradicts plur at
<* :SPR:LI(1):HEAD:NUMBER>.
yes
| ?-
```

The "failure explanation" tells us where the contradiction causing the unification failure is located. This location is described by means of a path expression, where (the dummy symbol) '`*`' represents the top level and terms of the form '`LI(`*N*`)`' are used as features to refer to the *N*th item of a list. '`<* :SPR:LI(1):HEAD:NUMBER>`' means the '`<* :HEAD:NUMBER>`' value of the first item of the '`<* :SPR>`' list.

Those tasks where the inactive edge is of the required kind, i.e. which do not fail, are listed by '`prtasks(`*A,B,C,*`ok)`'. (Such a task may still have failed to produce a new edge due to the

failure of a Prolog call prompted by a subsequent `?`-term in the action list.)

3.5 Listing grammar items

The lexical information associated with a certain word is printed by means of a `plex/1` call. The argument is the word in question.

```
| ?- plex(dog).
--------------- Entry: dog: 1 hit(s).
* sign #1
  HEAD: * n #2
    NUMBER: * sing #3
  SPR: ----- #4 ---<
      **** sign #5
         HEAD: * det #6
           NUMBER: ^3
    >--------
  SEM: <>dog #7


yes
| ?-
```

A list of the accessible lexical entries is printed if the predicate `words/0` is called.

```
| ?- words.
[dog:0,dogs:0,the:0,all:0,every:0,sleep:0,sleeps:1]

yes
| ?-
```

The number associated with each word specifies in which cycle of lexical rule application the entry was generated. The 0 words are defined directly in the grammar file (by word form or lexeme entries), while, for instance, '`sleeps:1`' means that '`sleeps`' was defined in the first cycle of lexical rule application.

The grammar rules are listed by means of '`prules`'. The single rule of the Sample Grammar, for instance, appears as follows:

```
| ?- prules.
--- RULE:
   LEFT: * sign #1
         HEAD: * head_type #2
   RIGHT:
   ! _166 #3
   ! * sign #4
     HEAD: ^2
     SPR: ----- #5 ---<
         ***^3
       >--------
   ?-call compute_semantics(...)


yes
| ?-
```

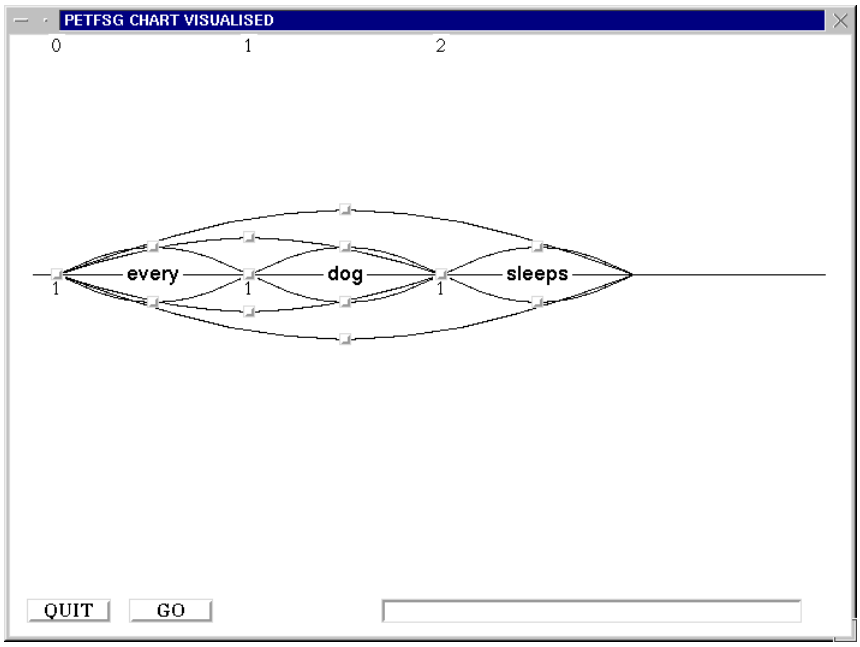
The rule is printed vertically: first the left-hand item, then the items of the right-hand list. An `?`-item is printed as '`?-call`' followed by the functor of the Prolog goal.

4 Graphical interface

The parser is equipped with a simple graphical interface, which produces a picture of the chart. The graphics appear in a separate window. The parser may also be used without the graphics.

The system enters the graphical mode when the predicate `show_chart/0` is called. A window showing the current chart will then appear.

In the graphical mode, the ordinary text-based interface to SICStus Prolog is used for textual output. The chart is drawn in the conventional manner: The words of the input string are printed along a horizontal line, and inactive edges are drawn above them, while active edges appear below the line, like this:



A button is placed on the crest of each edge. When mouse button 1 is released over the edge button, the content of the edge

is printed in the Prolog window (in the style of '`pri`' or '`pra`'). The buttons (between the words) on the chart nodes concern the active edges running from and to the node in question.

The number printed below a node gives the number of such edges (i.e. grammar rules invoked) at the node in question. They are listed if mouse button 1 is released over the node button.) (This is equivalent to the call '`pra(`*N*`,`*N*`)`', if *N* is the node number.) The node numbers appear above the node buttons.

Input to the system, when it is in the graphical mode, is entered through the entry widget. The following commands may be given:

- A string of the form '`?` *Number1 Number2 Number3*' (blanks separating) is interpreted as '`:-prtasks(`*Number1*`,`*Number2*`,`*Number3*`,fail)`').
- Other input is tokenized into a list of Prolog atoms and parsed. The chart picture is then redrawn. Each substring of characters without blanks counts as one token, blanks separating the tokens.

A command is executed when the "GO" button is hit. When the "QUIT" button is hit, the graphics window disappears and the system returns to the non-graphical mode.

5 How to download and run the PETFSG system

The URL '`http://stp.ling.uu.se/~matsd/code/petfsg/`' contains the PETFSG system and the illustration files. The system requires SICStus Prolog and Tcl/TK.

A convenient way of starting a session with the system and the Sample Grammar is to `consult` a file with the following content (found in the file `load_sample_grammar.pl`):

```
:- consult('petfsg.pl').
:- consult('sample_prolog.pl').
:- prepare_compilation.
:- compile_grammar('sample_grammar.pl').
```

Here, the file `petfsg.pl` is assumed to contain the PETFSG system. The file `sample_prolog.pl` contains definitions of predicates used by the grammar, and `sample_grammar.pl` is the Sample Grammar used for illustration here. The `prepare_compilation` call erases previous grammar items (if there are any), and declares the dynamic predicates required by the grammar compilation. Multiple `compile_grammar` calls are needed if the grammar is split into several files. The PETFSG file `petfsg.pl` only has to be consulted at the start of the session.

The Tcl/TK parameters which determine window size and chart diagram proportions are defined by predicates collected at the end of the file '`petfsg.pl`'. User preferences and/or screen properties are likely to motivate modifications.

References

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

Dahllöf, M., 1999, 'Implementing token-based minimal recursion semantics', *RUUL* (Reports from Uppsala University Department of Linguistics) **34** (this volume), 39–99.

Gazdar, G. and Mellish, C., 1989, *Natural Language Processing in PROLOG*, Wokingham, Addison-Wesley Publishing Company.

Pollard, C. and 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 1: The Sample Grammar

The Sample Grammar (in the file '`sample_grammar.pl`') covers a very small fragment of English and gives it a HPSG-inspired treatment. The only rule is a specifier-head rule. The subject-VP connection is assumed to be a special case of its application.

```
type_hierarchy([
   sign subsumes [],
   head_type subsumes [
      numbrd subsumes [
         n subsumes [],
         det subsumes []],
      v subsumes []],
   tense_type subsumes [
      present subsumes [],
      past subsumes []],
   number_type subsumes [
      sing subsumes [],
      plur subsumes []]]).

features(sign, [head:head_type,
        spr:list,
```

```
sem:prolog_term]).

features(numbrd, [number:number_type]).

features(v, [tense:tense_type]).

Mtr ===>  [!Spr,
   !Head,
   ?compute_semantics(Spr,Head,Mtr)] where
Head:spr === $[Spr],
Mtr:head === Head:head.

present_plural is_short_for
      § :head === v,
      § :head:tense === present,
      § :spr === $[Subj],
      Subj:head:number === plur.

noun_pattern is_pattern
   [p(id,
      (§ :head === n,
       § :head:number === sing)),
    p(affx(s),
      (§ :head === n,
       § :head:number === plur))].

dog of_pattern noun_pattern >>>
§ :head === n,
§ :head:number === Num,
§ :spr === $[Spr],
§ :sem === <> dog,
        Spr:head === det,
```

```
Spr:head:number === Num.

the >>>
§ :head === det,
§ :spr === $[],
§ :sem === <> the.

all >>>
§ :head === det,
§ :head:number === plur,
§ :spr === $[],
§ :sem === <> every.

every >>>
§ :head === det,
§ :head:number === sing,
§ :spr === $[],
§ :sem === <> every.

sleep >>>
§ === nfs(present_plural),
§ :spr === $[Subj],
§ :sem === <> sleep,
Subj:head === n,
Subj:spr === $[].

lexical_rule(A,B,affx(s)) where
        A:head === v,
A:head:tense === present,
A:spr === $[ASubj],
B:spr === $[BSubj],
ASubj:head:number === plur,
```

```
BSubj:head:number === sing,
A:head === B:head,
A:sem === B:sem,
BSubj:spr === ASubj:spr.
```

The grammar makes use of the following user-defined Prolog predicate (found in the file '`sample_prolog.pl`'):

```
compute_semantics(Spr,Head,Mtr):-
        Head:sem === <> Headsem,
        Spr:sem === <> Sprsem,
        Mtr:sem === <> (Sprsem@Headsem).
```

Here, the '`===`' predicate is used as the Prolog query in the grammar rule is supposed to succeed.

The specifier-head rule could also have been formulated in this way:

```
Mtr ===>  [!Spr,
           !Head]:-
        Head:spr === $[Spr],
        Mtr:head === Head:head,
        compute_semantics(Spr,Head,Mtr).
```

This formulation would be better from a computational point of view as the '`compute_semantics(Spr,Head,Mtr)`' call would only be made once, during compilation, and would result in structure sharing. This is more efficient and reflects the monotonic nature of this semantic constraint.

Appendix 2: 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 is dealt with by the Prolog system.

An error will produce a report of this kind:

```
E R R O R: Unknown type (plru).
--- Term in grammar, ending on line 70, was ignored
due to reported problem.
```

The grammar term producing this report was the following, where '`plru`' is an unknown, i.e. undeclared, type.

```
all >>>
§ :head === det,
§ :head:number === plru,
§ :spr === $[],
§ :sem === <> every.
```

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

```
F A I L U R E: plur contradicts sing at
<* :head:number>.
--- Term in grammar, ending on line 61, was ignored
due to reported problem.
```

This message is produced when the grammar encounters a word form entry containing the contradictory specifications, given the declarations of the Sample Grammar, '`§ :head:number === plur`' and '`§ :head:number === sing`'.

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

- `Unexpected expression (`*X*`)`: *X* is a Prolog term not expected to occur in a PETFSG grammar.
- `Additional type hierarchy`: An additional type hierarchy is found. There should be precisely one.
- `Type name already used (`*Type*`)`: Attempt to declare *Type* as the name of a type a second time.
- `Illegal feature declaration (`*Type*`)`: Attempt 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.
- `Feature name already used (`*Feat*`)`: Attempt to associate the already used feature name *Feat* with another type.
- `"list" is a reserved type name`: Attempt to define a type with the name '`list`', which has a reserved meaning in the system.
- `"prolog_term" is a reserved type name`: Attempt 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 a FS, is used a second time for that purpose.
- `Pattern name already used (`*Name*`)`: A name *Name*, already used to name an inflectional pattern, is used a second time for that purpose.

- `Unexpected kind of lexical shape (`*Lex*`)`: *Lex* is a non-atom appearing as the shape of a word form or lexeme.
- `Unexpected variable where type was expected`: Variable occurring where a type specification was expected.
- `Unexpected variable where feature name was expected`: A variable occurs in a path expression in a position where a feature name is required.
- `Unexpected variable in right-hand list of rule`: A variable is found in the right-hand list of a grammar rule. The elements in this list should be specified as having '`!`', '`+`', or '`?`' as their main functor.
- `Unexpected item in right-hand list of rule (`*A*`)`: Another kind of faulty item in right-hand list of rule (i.e. one which has not '`!`', '`+`', or '`?`' as its main functor).
- `Unknown type (`*Type*`)`: *Type* is not the name of a known type.
- `Unknown feature (`*Feat*`)`: Attempt to use *Feat* as a feature name while it is unknown as such.
- `Unknown morphological process (`*X*`)`: *X* is occurring in a lexeme entry or lexical rule as a morphological process, while being unknown as such.
- `Unknown fs name (`*Name*`)`: Attempt to access a FS by the name *Name*, which is not known to name a FS.
- `Unknown pattern (`*Name*`)`: Attempt to access an inflectional pattern by the name *Name*, which is not known to name an inflectional pattern.

- `Unknown entity, cannot evaluate (`*Val*`)`: The term *Val* does not represent a known value in the context of a '`===`' or '`denotes`' statement that is being evaluated.
- `Type does not allow feature (`*Feat*`)`: Attempt to associate feature name *Feat* with a FS of inappropriate type (in the context of a '`===`'-statement).
- `Wrong value for feature (`*Feat*`)`: Attempt to assign a value of the wrong type to a feature by the name *Feat* (in the context of a '`===`'-statement).
- `Failure:` *Type1* `contradicts` *Type2* `at` *Path*: Unification failure at *Path*, counted from the path given as the first argument to '`===`'. The main FS is represented as '`*`'. The types *Type1* and *Type2* are incompatible.
- `Failure: lists of different lengths at` *Path*: Unification failure at *Path* due to lists of different lengths.
- `Object description completely underspecified`: This error occurs when a word form , or lexeme description, or FS to be named is left completely underspecified.
- `Not a redefinable predicate (`*Term*`)`: *Term* does not represent a definition of `report_look_up/2`, `token_description/3`, or `fs_output_form/2`.

RUUL (Reports from Uppsala University Dept of Linguistics) **34**, 39–99.

Implementing token-based minimal recursion semantics

Mats Dahllöf
Department of Linguistics, Uppsala University
E-mail: `Mats.Dahllof@ling.uu.se`
September, 1999

1 Introduction

This report describes an implementation of the formal grammar and semantics (henceforth TBMRS) introduced in my article 'Token-Based Minimal Recursion Semantics: Davidson's Paratactic Treatment of Intensional Constructions in a Formal Syntax and Semantics' (Dahllöf 1999b). 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 aim is to illustrate the treatment of intensional constructions, and the grammar only covers toy fragments of other constructions.

The grammar as implemented here uses the system and formalism "Prolog-Embedding Typed Feature Structure Grammar", henceforth PETFSG (Dahllöf 1999a, this volume). This system uses typed feature structures in which Prolog-terms may be embedded. The rules allow Prolog calls to be made. Here, Prolog-terms are used to represent semantic objects. Prolog is

only used to append lists. Otherwise, the grammar relies on unification.

The main part of this report is a listing of the PETFSG representation of the TBMRS syntax and semantics, with some comments and explanations added. This presentation presupposes the linguistic and technical background given in Dahllöf (1999a, this volume) and (1999b).

The files of the implementation are available at the URL `http://stp.ling.uu.se/~matsd/code/tbmrs/`.

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*. Its only other subtype is `none`, which is used to negate the presence of a sign. This possibility is used to prevent expressions from being associated with heads as a pre- or postmodifiers. See below.

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

```
features(sign, [
   token: prolog_term,
   head:head_type,
   spr:list,
   comps:list,
   marking:marking_type,
   cstr:cstr_type,
   cont:cont]).
```

The features are used as described in Dahllöf (1999b), with the exception of `cstr`, whose value is the constituent structure associated with the expression in question. See code for details.

1.2 `cont`(ent) features and variables

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

```
features(cont, [
    lzt:list,
    wtop: prolog_term,
    hcons: prolog_term,
    varuse: prolog_term]).
```

The features `lzt` (liszt), `wtop` (working top), and `hcons` (handle constraints), are used as described in Dahllöf (1999b). The TBMRS variables and underspecified handles are represented as Prolog variables. The feature `varuse` carries a description of their use. Its value is a list of terms, each of which is of the form '`thing(`V`)`', '`event(`V`)`', or '`handle(`V`)`', V being the variable concerned. This information is used to improve readability. In the printed form, the TBMRS variables and handles appear as Prolog constants of letter-with-number form. The three kinds of expression are distinguished by the choice of letter, '`x`' for ordinary objectual "thing" variables, '`e`' for event variables, and '`h`' for handles. The `varuse` information originates from the lexicon. Its value on a non-lexical node is the union of the the `varuse` values on its daughters.

1.3 Token identifier assignment

The TBMRS grammar presupposes that the tokens are referred to by the grammatical descriptions. From a computational point of view, this means that each token must be assigned a unique identifier. This is done procedurally with the help of the predicate `token_description/3`, which 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 graphical form (an atom). The token identifier is the constant consisting of a 't' and the position number, produced by the predicate `token_identifier/2`, which is defined directly in Prolog (i.e. not in the PETFSG grammar file) by the clause:

```
token_identifier(X,Id):-
      name(X,Name),
      name(Id,[116|Name]).
```

This leads to the following definition of `token_description/3`:

```
redefine_predicate(
  (token_description(Desc,N,_):-
  (token_identifier(N,TokId),
   Desc:token === <> TokId))).
```

2 The grammar file

This section exhibits the entire grammar verbatim, with a few comments.

2.1 The type hierarchy

```
type_hierarchy([

   aops subsumes [       %%% absent or present sign
      sign subsumes [],  %%% present sign
      none subsumes []], %%% absent sign

   head_type subsumes [
```

```
      v subsumes [],
      determin subsumes [],
        nom subsumes [
            commonnoun subsumes [],
            perspron subsumes [],
            propernoun subsumes []],
         adj subsumes [],
         adv subsumes [],
         prep subsumes [],
         mark subsumes []],

   mod_type subsumes [],

   cont subsumes [],

   semrel subsumes [
      semrel_qua subsumes [],
      semrel_pred subsumes [
         semrel_rel subsumes [],
         semrel_pr subsumes []]],

   boolean subsumes [
      no subsumes [],
      yes subsumes []],

   cstr_type subsumes [
      leaf subsumes [],
      hdstr subsumes [
            hd_mod subsumes [],
            hd_mark subsumes [],
            hd_sp subsumes [],
            hd_comps subsumes []],
```

```
   coordstr subsumes []],

marking_type subsumes [
   unmarked subsumes [],
   marked subsumes [
      to_marked subsumes [],
      that_marked subsumes []]],

number_type subsumes [
   sing subsumes [],
   plur subsumes []],

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

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

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

vform_type subsumes [
   finite subsumes [
         present subsumes [],
         preterit subsumes []],
   infinite subsumes [
```

```
            infinitive subsumes [],
            supine subsumes [[]]]]).
```

2.2 The feature declarations

```
features(sign, [
   token:prolog_term,
   head:head_type,
   spr:list,
   comps:list,
   marking:marking_type,
   cstr:cstr_type,
   cont:cont]).

features(cont, [
    lzt:list,
    wtop:prolog_term,
    hcons:prolog_term,
    varuse:prolog_term]).
```

As mentioned in section 1.2, the feature `varuse` contains a list that characterizes the use of Prolog variables as representation language variables.

```
features(semrel, [
    handle:prolog_term,
    lo:prolog_term]).

features(semrel_qua, [
    bv:prolog_term,
    rs:prolog_term,
    sc:prolog_term]).
```

```
features(semrel_pred, [
    an:prolog_term]).

features(semrel_pr, [
    in:prolog_term]).

features(semrel_rel, [
    ev:prolog_term,
    su:prolog_term,
    c1:prolog_term]).

features(head_type, [
    hdtoken:prolog_term,
    inter:semrel,
    mod:mod_type,
    spec:sign]).

features(nom, [
    num:number_type,
    def:def_type,
    case:case_type,
    pers:pers_type]).

features(v, [
    vform:vform_type]).

features(mod_type, [
    pre:aops,      %%% as premodifier
    post:aops]).   %%% as postmodifier

features(hdstr, [
    hd_first:boolean,
```

```
    hd_dtr:sign]).

features(hd_sp, [
    spr_dtrs:list]).

features(hd_comps, [
    comp_dtrs:list]).

features(hd_mod, [
    mod_dtr:sign]).

features(hd_mark, [
    mark_dtr:sign]).

features(coordstr, [
    left_item:sign,
    coordinator:sign,
    right_item:sign]).
```

2.3 Token definition

```
redefine_predicate(
  (token_description(Desc,N,_):-
  (token_identifier(N,TokId),
   Desc:token === <>TokId))).
```

2.4 The grammar rules

2.4.1 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 per complement

is required. The append operations compute the mother's `lzt`, `varuse`, and `hcons` values.

```
Mtr ===>  [!Head,!Comp,
           ?append(HdLZ,CmpLZ,MtrLZ),
           ?append(VU1,VU2,VU),
           ?append(HdHC,CmpHC,MtrHC)]
                                    where
   Mtr:head === Head:head,
   Head:comps === $[Comp|Comps],
   Mtr:comps === $Comps,
   Mtr:spr === Head:spr,
   Mtr:marking === unmarked,

   Mtr:cstr:hd_first === yes,
   Mtr:cstr:hd_dtr === Head,
   Mtr:cstr:comp_dtrs === $[Comp],

   Mtr:cont:wtop === Head:cont:wtop,

   Mtr:cont:lzt === $MtrLZ,
   Head:cont:lzt === $HdLZ,
   Comp:cont:lzt === $CmpLZ,
   Mtr:cont:hcons === <>MtrHC,
   Head:cont:hcons === <>HdHC,
   Comp:cont:hcons === <>CmpHC,
   Mtr:cont:varuse === <>VU,
   Head:cont:varuse === <>VU1,
   Comp:cont:varuse === <>VU2,

   Head:token === <>HdTok,
   Comp:token === <>CmpTok,
```

```
   Mtr:token === <>(HdTok+CmpTok).
```

The + operator represents physical aggregation.

2.4.2 The specifier-head rule

```
Mtr ===>  [!Spr, !Head,
           ?append(HdHC,SprHC,MtrHC),
           ?append(VU1,VU2,VU),
           ?append(SprLZ,HdLZ,MtrLZ)] where
   Mtr:head === Head:head,
   Head:spr === $[Spr],
   Spr:head:spec === Head,
   Head:comps === $[],
   Mtr:comps === Head:comps,
   Mtr:spr === $[],
   Mtr:marking === unmarked,

   Mtr:cstr:hd_first === no,
   Mtr:cstr:hd_dtr === Head,
   Mtr:cstr:spr_dtrs === $[Spr],

   Mtr:cont:lzt === $MtrLZ,
   Head:cont:lzt === $HdLZ,
   Spr:cont:lzt === $SprLZ,
   Mtr:cont:hcons === <>MtrHC,
   Head:cont:hcons === <>HdHC,
   Spr:cont:hcons === <>SprHC,
   Mtr:cont:varuse === <>VU,
   Spr:cont:varuse === <>VU1,
   Head:cont:varuse === <>VU2,

   Head:token === <>HdTok,
```

```
   Spr:token === <>SprTok,
   Mtr:token === <>(SprTok+HdTok).
```

The equation `Spr:head:spec === Head` implements Pollard's and Sag's (1994: 51) "Spec Principle". The `wtop` value is left an open issue.

2.4.3 The premodifier-head rule

```
Mtr ===>  [!Mod,!Head,
           ?append(ModLZ,HdLZ,MtrLZ),
           ?append(VU1,VU2,VU),
           ?append(HdHC,ModHC,MtrHC)] where
   Mtr:head === Head:head,
   Mod:head:mod:pre === Head,
   Mod:comps === $[],
   Mtr:comps === Head:comps,
   Mtr:spr === Head:spr,
   Mtr:marking === unmarked,

   Mtr:cstr:hd_first === no,
   Mtr:cstr:hd_dtr === Head,
   Mtr:cstr:mod_dtr === Mod,

   Mtr:cont:wtop === Mod:cont:wtop,

   Mtr:cont:lzt === $MtrLZ,
   Head:cont:lzt === $HdLZ,
   Mod:cont:lzt === $ModLZ,
   Mtr:cont:hcons === <>MtrHC,
   Head:cont:hcons === <>HdHC,
   Mod:cont:hcons === <>ModHC,
   Mtr:cont:varuse === <>VU,
```

```
   Mod:cont:varuse === <>VU1,
   Head:cont:varuse === <>VU2,

   Head:token === <>HdTok,
   Mod:token === <>ModTok,
   Mtr:token === <>(ModTok+HdTok).
```

2.4.4 The marker-head rule

```
Mtr ===>  [!Mark,!Head] where
   Mtr:head === Head:head,
   Mark:head === mark,
   Mark:head:spec === Head,
   Mark:marking === marked,
   Mtr:marking === Mark:marking,
   Mtr:comps === Head:comps,
   Mtr:spr === Head:spr,

   Mtr:cstr:hd_first === no,
   Mtr:cstr:hd_dtr === Head,
   Mtr:cstr:mark_dtr === Mark,

   Mtr:cont === Head:cont,

   Head:token === <>HdTok,
   Mark:token === <>MarkTok,
   Mtr:token === <>(MarkTok+HdTok).
```

The equation `Mark:head:spec === Head` implements Pollard's and Sag's (1994: 51) "Spec Principle".

2.5 Lexical information

2.5.1 General definitions

```
lex_general is_short_for
   § :head:hdtoken === § :token,
   § :cont:wtop === § :token,
   § :cont:lzt === $[First_Semrel|_],
   First_Semrel:handle === § :token.
```

This applies to all basic lexical entries.

```
np_sub is_short_for  %%% subject form np:s
   § :head === nom,
   § :head:case === subc,
   § :spr === $[],
   § :comps === $[].

np_obj is_short_for  %%% object form np:s
   § :head === nom,
   § :head:case === objc,
   § :spr === $[],
   § :comps === $[].

infinitival_phrase is_short_for
   § :head === v,
   § :head:vform === infinitive,
   § :marking === to_marked,
   § :spr === $[],
   § :comps === $[].

no_mod is_short_for
   § :head:mod:pre === none,
   § :head:mod:post === none.
```

2.5.2 Proper nouns

```
proper_noun is_short_for
   § === nfs(lex_general),
   § :head === propernoun,
   § :head:num === sing,
   § :head:pers === p3,
   § :spr === $[],
   § :comps === $[],
   § === nfs(no_mod),
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel:bv === <>Var,
   Rel:rs === <>nil,
   Rel:sc === <>SC,
   § :head:inter:bv === Rel:bv,
   § :cont:wtop === <>Tok,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(Var),handle(SC)].

galileo >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Galileo').
```

2.6 Personal pronouns

```
personal_pronoun is_short_for
   § === nfs(lex_general),
   § :head === perspron,
   § :head:num === sing,
   § :spr === $[],
```

```
   § :comps === $[],
   § === nfs(no_mod),
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel:bv === <>Var,
   Rel:rs === <>nil,
   Rel:sc === <>SC,
   § :head:inter:bv === Rel:bv,
   § :cont:wtop === § :token,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(Var),handle(SC)].

he >>>
   §  === nfs(personal_pronoun),
   § :head:pers === p3,
   § :head:num === sing,
   § :cont:lzt === $[Rel],
   Rel:lo === <>perspro('he').
```

2.6.1 Determiners

```
determiner is_short_for
   § === nfs(lex_general),
   § :head === determin,
   § :spr === $[],
   § :comps === $[],
   § === nfs(no_mod),
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel:bv === <>Var,
   Rel:rs === § :head:spec:cont:wtop,
   Rel:rs === <>Restr,
```

```
   Rel:sc === <>Scope,
   § :head:inter:bv === Rel:bv,
   § :head:inter:sc === Rel:sc,
   § :head:inter:rs === Rel:rs,
   § :cont:wtop === <>Tok,
   § :cont:varuse === <>[thing(Var),handle(Restr),
                         handle(Scope)].

many >>>
   §  === nfs(determiner),
   § :head:spec:head:num === plur,
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(many),
   § :cont:hcons === <>[].
```

2.6.2 Common noun morphology

```
noun_pattern is_pattern
   [p(id,
      § :head:num === sing),
    p(affx(s),
      § :head:num === plur)].
```

2.6.3 Common nouns requiring a specifier

```
common_noun_spr is_short_for
   § === nfs(lex_general),
   § :head === commonnoun,
   § :head:num === _,
   § :head:pers === p3,
   § :spr === $[Spr],
   Spr:head === determin,
   § :comps === $[],
```

```
   § === nfs(no_mod),
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel:in === Spr:head:inter:bv,
   § :head:inter:bv === Rel:in,
   § :head:inter:sc === Spr:head:inter:sc,
   § :cont:wtop === § :token,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[].

book of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(book).
```

2.6.4 Verb analysis: general aspects

```
verb_general is_short_for
   § === nfs(lex_general),
   § :head === v,
   § === nfs(no_mod),
   § :marking === unmarked,
   § :cont:wtop === § :token.
```

2.6.5 Verb form abbreviations

```
present_third_singular is_short_for
   § :spr === $[Subj],
   § :head:vform === present,
   Subj === nfs(np_sub),
   Subj:head:pers === p3,
   Subj:head:num === sing,
   Subj:head:inter:bv === § :head:inter:su.
```

```
present_third_plural is_short_for
   § :spr === $[Subj],
   § :head:vform === present,
   Subj === nfs(np_sub),
   Subj:head:pers === p3,
   Subj:head:num === plur,
   Subj:head:inter:bv === § :head:inter:su.

preteritum is_short_for
   § :spr === $[Subj],
   § :head:vform === preterit,
   Subj === nfs(np_sub),
   Subj:head:inter:bv === § :head:inter:su.

infinitive_form is_short_for
   § :head:vform === infinitive,
   § :spr === $[],
   Subj:head:inter:bv === § :head:inter:su.
```

2.6.6 Intransitive verbs

```
verb_intransitive is_short_for
   § === nfs(verb_general),
   § :comps === $[],
   § :cont:lzt === $[Rel],
   Rel:ev === <>VarE,
   Rel:su === <>VarS,
   § :head:inter:ev === Rel:ev,
   § :head:inter:su === Rel:su,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(VarS),event(VarE)].
```

2.6.7 Ordinary transitive verbs

```
verb_transitive is_short_for
   § === nfs(verb_general),
   § :comps === $[Comp],  Comp === nfs(np_obj),
   Comp:head:inter:bv === Rel:c1,
   § :cont:lzt === $[Rel],
   Rel:ev === <>VarE,
   Rel:su === <>VarS,
   Rel:c1 === <>VarC,
   § :head:inter:ev === Rel:ev,
   § :head:inter:su === Rel:su,
   § :head:inter:c1 === Rel:c1,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(VarS),thing(VarC),
                         event(VarE)].

reads >>>
   §  === nfs(verb_transitive),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(read).
```

2.6.8 Paratactic verbs

```
verb_paratactic is_short_for
   § === nfs(verb_general),
   § :comps === $[Comp],
   Comp:head === v,
   Comp:head:hdtoken === <>HdTok,
   Comp:comps === $[],
   Comp:spr === $[],
   § :cont:lzt === $[Rel],
```

```
   Rel:ev === <>VarE,
   Rel:su === <>VarS,
   Rel:c1 === <>ltop(HdTok),
   § :head:inter:ev === Rel:ev,
   § :head:inter:su === Rel:su,
   § :head:inter:c1 === Rel:c1,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(VarS),handle(HdTok),
                         event(VarE)].

said >>>
   §  === nfs(verb_paratactic),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(said).
```

2.6.9 Infinitival attitude verbs

```
verb_infinitival_attitude is_short_for
   § === nfs(verb_general),
   § :comps === $[Comp],
   Comp === nfs(infinitival_phrase),
   Comp:head:hdtoken === <>HdTok,
   § :cont:lzt === $[Rel],
   Rel:su === <>VarS,
   Rel:ev === <>VarE,
   Rel:c1 === <>ltop(HdTok),
   § :head:inter:ev === Rel:ev,
   § :head:inter:su === Comp:head:inter:su,
   § :head:inter:su === Rel:su,
   § :head:inter:c1 === Rel:c1,
   § :cont:hcons === <>[],
```

```
   § :cont:varuse === <>[event(VarE),thing(VarS),
                        handle(HdTok),
                        handle(VarH)].

intends >>>
   §  === nfs(verb_infinitival_attitude),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(intends).
```

2.6.10 Attributary verbs

```
verb_attributary is_short_for
   § === nfs(verb_general),
   § :comps === $[Comp],
   Comp === nfs(np_obj),
   Comp:head:hdtoken === <>HdTok,
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(Verb),
   Rel:ev === <>VarE,
   Rel:su === <>VarS,
   Rel:c1 === <>ltop(HdTok),
   § :cont:hcons === <>[],
   § :head:inter:ev === Rel:ev,
   § :head:inter:su === Rel:su,
   § :cont:varuse === <>[handle(HdTok),thing(VarS),
                        event(VarE)].

seeks >>>
   §  === nfs(verb_attributary),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel|_],
```

```
   Rel:lo === <>rel(seeks).
```

2.6.11 Sentence adverbials

```
sent_adv is_short_for
   § === nfs(lex_general),
   § :head === adv,
   § :spr === $[],
   § :comps === $[],
   § :head:mod:pre:head === v,
   § :head:mod:pre:cstr === leaf,
   § :head:mod:pre:head:hdtoken === <>VHDT,
   § :head:mod:pre:marking === unmarked,
   § :head:mod:post === none,
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel :in === <>ltop(VHDT),
   § :cont:wtop === § :token,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[handle(VHDT)].

consequently >>>
   §  === nfs(sent_adv),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(consequently).
```

2.6.12 Subject-relative adverbs

```
sent_adv_subj is_short_for
   § === nfs(lex_general),
   § :head === adv,
   § :spr === $[],
   § :comps === $[],
```

```
   § :head:mod:pre:head === v,
   § :head:mod:pre:head:hdtoken === <>VHDT,
   § :head:mod:pre:cstr === leaf,
   § :head:mod:pre:marking === unmarked,
   § :head:mod:post === none,
   § :marking === unmarked,
   § :cont:lzt === $[Rel],
   Rel:su === § :head:mod:pre:head:inter:su,
   Rel:ev === § :head:mod:pre:head:inter:ev,
   Rel:c1 === <>ltop(VHDT),
   § :cont:wtop === § :token,
   Rel:su === <>VarS,
   Rel:ev === <>VarE,
   § :cont:hcons === <>[],
   § :cont:varuse === <>[thing(VarS),event(VarE)].

intentionally >>>
   §  === nfs(sent_adv_subj),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(intentionally).
```

Produces a singular form identical to the stem and a plural form with s-ending.

2.6.13 Markers

```
that >>>
   § :head === mark,
   § :head:spec === V,
   § :marking === that_marked,
   § :spr === $[],
   § :comps === $[],
   § === nfs(no_mod),
```

```
   V :head === v,
   V :marking === unmarked,
   V :spr === $[],
   V :comps === $[].

to >>>
   § :head === mark,
   § :head:spec === V,
   § :marking === to_marked,
   § :spr === $[],
   § :comps === $[],
   § === nfs(no_mod),
   V :head === v,
   V :head:vform === infinitive,
   V :marking === unmarked,
   V :comps === $[].
```

2.7 Additional entries

2.7.1 Determiners

```
a >>>
   §  === nfs(determiner),
   § :head:spec:head:num === sing,
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(a),
   § :cont:hcons === <>[].

each >>>
   §  === nfs(determiner),
   § :head:spec:head:num === sing,
   § :cont:lzt === $[Rel],
```

```
   Rel:lo === <>det(each),
   § :cont:hcons === <>[].

most >>>
   §  === nfs(determiner),
   § :head:spec:head:num === plur,
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(most),
   § :cont:hcons === <>[].

the >>>
   §  === nfs(determiner),
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(the),
   § :cont:hcons === <>[].

three >>>
   §  === nfs(determiner),
   § :head:spec:head:num === plur,
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(three),
   § :cont:hcons === <>[].

two >>>
   §  === nfs(determiner),
   § :head:spec:head:num === plur,
   § :cont:lzt === $[Rel],
   Rel:lo === <>det(two),
   § :cont:hcons === <>[].

catiline >>>
   §  === nfs(proper_noun),
```

```
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Catiline').

cicero >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Cicero').

davidson >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Davidson').

john >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('John').

mary >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Mary').

oedipus >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Oedipus').

tully >>>
   §  === nfs(proper_noun),
   § :cont:lzt === $[Rel],
   Rel:lo === <>name('Tully').
```

2.7.2 Common nouns

```
apple of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(apple).

boy of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(boy).

driver of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(driver).

file of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(file).

groundhog of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(groundhog).

student of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
```

```
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(student).

teacher of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(teacher).

unicorn of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(unicorn).

woodchuck of_pattern noun_pattern >>>
   §  === nfs(common_noun_spr),
   § :cont:lzt === $[Rel],
   Rel:lo === <>noun(woodchuck).
```

2.7.3 Intransitive verb forms

```
disappeared >>>
   §  === nfs(verb_intransitive),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(disappeared).

sleep >>>
   §  === nfs(verb_intransitive),
   §  === nfs(present_third_plural),
   § :cont:lzt === $[Rel],
```

```
   Rel:lo === <>rel(sleep).

sleeps >>>
   §  === nfs(verb_intransitive),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(sleep).
```

2.7.4 Extensional transitive verb forms

```
ate >>>
   §  === nfs(verb_transitive),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(ate).

buy >>>
   §  === nfs(verb_transitive),
   §  === nfs(infinitive_form),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(read).

denounced >>>
   §  === nfs(verb_transitive),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(denounced).

erased >>>
   §  === nfs(verb_transitive),
```

```
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(erased).

find >>>
   §  === nfs(verb_transitive),
   §  === nfs(infinitive_form),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(find).

read >>>
   §  === nfs(verb_transitive),
   §  === nfs(present_third_plural),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(read).

read >>>
   §  === nfs(verb_transitive),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(read).

read >>>
   §  === nfs(verb_transitive),
   §  === nfs(infinitive_form),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(read).

saw >>>
   §  === nfs(verb_transitive),
   §  === nfs(preteritum),
   § :cont:lzt === $[Rel],
```

```
   Rel:lo === <>rel(saw).
```

2.7.5 Intensional verb forms

```
believes >>>
   §  === nfs(verb_paratactic),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(believes).

says >>>
   §  === nfs(verb_paratactic),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(says).

tries >>>
   §  === nfs(verb_infinitival_attitude),
   §  === nfs(present_third_singular),
   § :cont:lzt === $[Rel],
   Rel:lo === <>rel(tries).
```

2.7.6 Adverbs

```
arguably >>>
   §  === nfs(sent_adv),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(arguably).
```

```
probably >>>
   §  === nfs(sent_adv),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(probably).

deliberately >>>
   §  === nfs(sent_adv_subj),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(deliberately).

unintentionally >>>
   §  === nfs(sent_adv_subj),
   § :cont:lzt === $[Rel],
   Rel:lo === <>adv(unintentionally).
```

2.8 FS output selection

```
redefine_predicate(
  (fs_output_form(all,ALL,ALL):-
      (ALL:'cont':varuse === <>Varuse,
       select_readable_vars(Varuse,1,_,1,_,1,_)))).

redefine_predicate(
  (fs_output_form(lzt,ALL,SEL):-
      (ALL:'cont':lzt === SEL:'cont':lzt,
       ALL:'cont':varuse === <>Varuse,
       select_readable_vars(Varuse,1,_,1,_,1,_)))).
```

The predicate `fs_output_form/3` defines the output form of the analyses. Here, `all`-type output gives the entire analyses, whereas `lzt`-type output is restricted to the `lzt` value. In both cases, the predicate `select_readable_vars/3` unifies the

Prolog variables representing the TBMRS variables with more readable constants, as described in section 1.2.

3 Auxiliary Prolog predicates

The grammar makes use of three auxiliary Prolog predicates: `token_identifier/2` (see section 1.3), `morphology/3` (see below and Dahllöf 1999a, this volume, section 2.8.1), and `select_readable_vars/7` (see section 1.2).

3.1 Morphology

The following Prolog clauses define the morphological processes used by the grammar:

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

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

4 Analyses of words and sentences

The following analyses are as produced by the present grammar. The predicate `pp/2` has been used for parsing. The examples are the ones from Dahllöf (1999b), taken in the order in which they appear there. In a few places, lines have been manually broken.

```
----- [john,says,that,mary,saw,two,groundhogs]
```

```
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
         **** semrel_qua #4
            HANDLE: <>t1 #5
            LO: <>name(John) #6
            BV: <>x1 #7
            RS: <>nil #8
            SC: <>h1 #9
         **** semrel_rel #10
            HANDLE: <>t2 #11
            LO: <>rel(says) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t5) #14
         **** semrel_qua #15
            HANDLE: <>t4 #16
            LO: <>name(Mary) #17
            BV: <>x2 #18
            RS: <>nil ^8
            SC: <>h2 #19
         **** semrel_rel #20
            HANDLE: <>t5 #21
            LO: <>rel(saw) #22
            EV: <>e2 #23
            SU: <>x2 ^18
            C1: <>x3 #24
         **** semrel_qua #25
            HANDLE: <>t6 #26
            LO: <>det(two) #27
            BV: <>x3 ^24
```

```
          RS: <>t7 #28
          SC: <>h3 #29
        **** semrel_pr #30
          HANDLE: <>t7 ^28
          LO: <>noun(groundhog) #31
          IN: <>x3 ^24
      >--------
----- [two,boys,ate,three,apples]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
          HANDLE: <>t1 #5
          LO: <>det(two) #6
          BV: <>x1 #7
          RS: <>t2 #8
          SC: <>h1 #9
        **** semrel_pr #10
          HANDLE: <>t2 ^8
          LO: <>noun(boy) #11
          IN: <>x1 ^7
        **** semrel_rel #12
          HANDLE: <>t3 #13
          LO: <>rel(ate) #14
          EV: <>e1 #15
          SU: <>x1 ^7
          C1: <>x2 #16
        **** semrel_qua #17
          HANDLE: <>t4 #18
          LO: <>det(three) #19
          BV: <>x2 ^16
```

```
            RS: <>t5 #20
            SC: <>h2 #21
         **** semrel_pr #22
            HANDLE: <>t5 ^20
            LO: <>noun(apple) #23
            IN: <>x2 ^16
      >--------
----- [mary,says,that,two,books,disappeared]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
         **** semrel_qua #4
            HANDLE: <>t1 #5
            LO: <>name(Mary) #6
            BV: <>x1 #7
            RS: <>nil #8
            SC: <>h1 #9
         **** semrel_rel #10
            HANDLE: <>t2 #11
            LO: <>rel(says) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t6) #14
         **** semrel_qua #15
            HANDLE: <>t4 #16
            LO: <>det(two) #17
            BV: <>x2 #18
            RS: <>t5 #19
            SC: <>h2 #20
         **** semrel_pr #21
            HANDLE: <>t5 ^19
```

```
           LO: <>noun(book) #22
           IN: <>x2 ^18
         **** semrel_rel #23
           HANDLE: <>t6 #24
           LO: <>rel(disappeared) #25
           EV: <>e2 #26
           SU: <>x2 ^18
      >--------
----- [many,books]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <> (t1+t2) #2
  HEAD: * commonnoun #3
    HDTOKEN: <>t2 #4
    INTER: * semrel_qua #5
      BV: <>x1 #6
      SC: <>h1 #7
    MOD: * mod_type #8
      PRE: * none #9
      POST: * none #10
    NUM: * plur #11
    PERS: * p3 #12
  SPR: []
  COMPS: []
  MARKING: * unmarked #13
  CSTR: * hd_sp #14
    HD_FIRST: * no #15
    HD_DTR: * sign #16
      TOKEN: <>t2 ^4
      HEAD:  ^3
      SPR: ----- #17 ---<
          **** sign #18
```

```
        TOKEN: <>t1 #19
        HEAD: * determin #20
          HDTOKEN: <>t1 ^19
          INTER: * semrel_qua #21
            BV: <>x1 ^6
            RS: <>t2 ^4
            SC: <>h1 ^7
          MOD: * mod_type #22
            PRE: * none #23
            POST: * none #24
          SPEC:  ^16
        SPR: []
        COMPS: []
        MARKING: * unmarked #25
        CONT: * cont #26
          LZT: ----- #27 ---<
              **** semrel_qua #28
                 HANDLE: <>t1 ^19
                 LO: <>det(many) #29
                 BV: <>x1 ^6
                 RS: <>t2 ^4
                 SC: <>h1 ^7
            >--------
          WTOP: <>t1 ^19
          HCONS: <>[] #30
          VARUSE: <>[thing(x1),handle(t2),
                     handle(h1)] #31
    >--------
  COMPS: []
  MARKING: * unmarked #32
  CONT: * cont #33
    LZT: ----- #34 ---<
```

```
            **** semrel_pr #35
               HANDLE: <>t2 ^4
               LO: <>noun(book) #36
               IN: <>x1 ^6
             >--------
          WTOP: <>t2 ^4
          HCONS: <>[] ^30
          VARUSE: <>[] ^30
      SPR_DTRS:  ^17
    CONT: * cont #37
      LZT: ----- #38 ---<
          *** ^28
          *** ^35
        >--------
      HCONS: <>[] ^30
      VARUSE: <>[thing(x1),handle(t2),
                handle(h1)] ^31
----- [many]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * determin #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_qua #4
      BV: <>x1 #5
      RS: <>h1 #6
      SC: <>h2 #7
    MOD: * mod_type #8
      PRE: * none #9
      POST: * none #10
    SPEC: * sign #11
      HEAD: * nom #12
```

```
        NUM: * plur #13
      CONT: * cont #14
        WTOP: <>h1 ^6
  SPR: []
  COMPS: []
  MARKING: * unmarked #15
  CONT: * cont #16
    LZT: ----- #17 ---<
        **** semrel_qua #18
           HANDLE: <>t1 ^2
           LO: <>det(many) #19
           BV: <>x1 ^5
           RS: <>h1 ^6
           SC: <>h2 ^7
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #20
    VARUSE: <>[thing(x1),handle(h1),
               handle(h2)] #21
----- [books]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * commonnoun #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_qua #4
      BV: <>_12928 #5
      SC: <>_12925 #6
    MOD: * mod_type #7
      PRE: * none #8
      POST: * none #9
    NUM: * plur #10
```

```
    PERS: * p3 #11
  SPR: ----- #12 ---<
      **** sign #13
         HEAD: * determin #14
           INTER: * semrel_qua #15
             BV: <>_12928 ^5
             SC: <>_12925 ^6
    >--------
  COMPS: []
  MARKING: * unmarked #16
  CONT: * cont #17
    LZT: ----- #18 ---<
        **** semrel_pr #19
           HANDLE: <>t1 ^2
           LO: <>noun(book) #20
           IN: <>_12928 ^5
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #21
    VARUSE: <>[] ^21
----- [galileo]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * propernoun #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_qua #4
      BV: <>x1 #5
    MOD: * mod_type #6
      PRE: * none #7
      POST: * none #8
    NUM: * sing #9
```

```
    PERS: * p3 #10
  SPR: []
  COMPS: []
  MARKING: * unmarked #11
  CONT: * cont #12
    LZT: ----- #13 ---<
        **** semrel_qua #14
           HANDLE: <>t1 ^2
           LO: <>name(Galileo) #15
           BV: <>x1 ^5
           RS: <>nil #16
           SC: <>h1 #17
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #18
    VARUSE: <>[thing(x1),handle(h1)] #19
----- [reads]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * v #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_rel #4
      EV: <>e1 #5
      SU: <>x1 #6
      C1: <>x2 #7
    MOD: * mod_type #8
      PRE: * none #9
      POST: * none #10
    VFORM: * present #11
  SPR: ----- #12 ---<
      **** sign #13
```

```
        HEAD: * nom #14
          INTER: * semrel_qua #15
            BV: <>x1 ^6
          NUM: * sing #16
          CASE: * subc #17
          PERS: * p3 #18
        SPR: []
        COMPS: []
    >--------
  COMPS: ----- #19 ---<
      **** sign #20
         HEAD: * nom #21
           INTER: * semrel_qua #22
             BV: <>x2 ^7
           CASE: * objc #23
         SPR: []
         COMPS: []
    >--------
  MARKING: * unmarked #24
  CONT: * cont #25
    LZT: ----- #26 ---<
        **** semrel_rel #27
           HANDLE: <>t1 ^2
           LO: <>rel(read) #28
           EV: <>e1 ^5
           SU: <>x1 ^6
           C1: <>x2 ^7
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #29
    VARUSE: <>[thing(x1),thing(x2),event(e1)] #30
----- [galileo,says,that,he,reads,two,books]
```

```
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
         **** semrel_qua #4
            HANDLE: <>t1 #5
            LO: <>name(Galileo) #6
            BV: <>x1 #7
            RS: <>nil #8
            SC: <>h1 #9
         **** semrel_rel #10
            HANDLE: <>t2 #11
            LO: <>rel(says) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t5) #14
         **** semrel_qua #15
            HANDLE: <>t4 #16
            LO: <>perspro(he) #17
            BV: <>x2 #18
            RS: <>nil ^8
            SC: <>h2 #19
         **** semrel_rel #20
            HANDLE: <>t5 #21
            LO: <>rel(read) #22
            EV: <>e2 #23
            SU: <>x2 ^18
            C1: <>x3 #24
         **** semrel_qua #25
            HANDLE: <>t6 #26
            LO: <>det(two) #27
            BV: <>x3 ^24
```

```
          RS: <>t7 #28
          SC: <>h3 #29
        **** semrel_pr #30
          HANDLE: <>t7 ^28
          LO: <>noun(book) #31
          IN: <>x3 ^24
      >--------
----- [says]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * v #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_rel #4
      EV: <>e1 #5
      SU: <>x1 #6
      C1: <>ltop(h1) #7
    MOD: * mod_type #8
      PRE: * none #9
      POST: * none #10
    VFORM: * present #11
  SPR: ----- #12 ---<
      **** sign #13
        HEAD: * nom #14
          INTER: * semrel_qua #15
            BV: <>x1 ^6
          NUM: * sing #16
          CASE: * subc #17
          PERS: * p3 #18
        SPR: []
        COMPS: []
    >--------
```

```
  COMPS: ----- #19 ---<
      **** sign #20
         HEAD: * v #21
           HDTOKEN: <>h1 #22
         SPR: []
         COMPS: []
    >--------
  MARKING: * unmarked #23
  CONT: * cont #24
    LZT: ----- #25 ---<
        **** semrel_rel #26
           HANDLE: <>t1 ^2
           LO: <>rel(says) #27
           EV: <>e1 ^5
           SU: <>x1 ^6
           C1: <>ltop(h1) ^7
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #28
    VARUSE: <>[thing(x1),handle(h1),event(e1)] #29
----- [john,intends,to,read,a,book]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>name(John) #6
           BV: <>x1 #7
           RS: <>nil #8
           SC: <>h1 #9
        **** semrel_rel #10
```

```
            HANDLE: <>t2 #11
            LO: <>rel(intends) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t4) #14
          **** semrel_rel #15
            HANDLE: <>t4 #16
            LO: <>rel(read) #17
            EV: <>e2 #18
            SU: <>x1 ^7
            C1: <>x2 #19
          **** semrel_qua #20
            HANDLE: <>t5 #21
            LO: <>det(a) #22
            BV: <>x2 ^19
            RS: <>t6 #23
            SC: <>h3 #24
          **** semrel_pr #25
            HANDLE: <>t6 ^23
            LO: <>noun(book) #26
            IN: <>x2 ^19
        >--------
----- [intends]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * v #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_rel #4
      EV: <>e1 #5
      SU: <>x1 #6
      C1: <>ltop(h1) #7
```

```
  MOD: * mod_type #8
    PRE: * none #9
    POST: * none #10
  VFORM: * present #11
SPR: ----- #12 ---<
    **** sign #13
       HEAD: * nom #14
         INTER: * semrel_qua #15
           BV: <>x1 ^6
         NUM: * sing #16
         CASE: * subc #17
         PERS: * p3 #18
       SPR: []
       COMPS: []
  >--------
COMPS: ----- #19 ---<
    **** sign #20
       HEAD: * v #21
         HDTOKEN: <>h1 #22
         INTER: * semrel_rel #23
           SU: <>x1 ^6
         VFORM: * infinitive #24
       SPR: []
       COMPS: []
       MARKING: * to_marked #25
  >--------
MARKING: * unmarked #26
CONT: * cont #27
  LZT: ----- #28 ---<
      **** semrel_rel #29
         HANDLE: <>t1 ^2
         LO: <>rel(intends) #30
```

```
          EV: <>e1 ^5
          SU: <>x1 ^6
          C1: <>ltop(h1) ^7
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #31
    VARUSE: <>[event(e1),thing(x1),
               handle(h1),handle(h2)] #32
----- [galileo,consequently,read,two,books]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>name(Galileo) #6
           BV: <>x1 #7
           RS: <>nil #8
           SC: <>h1 #9
        **** semrel_pr #10
           HANDLE: <>t2 #11
           LO: <>adv(consequently) #12
           IN: <>ltop(t3) #13
        **** semrel_rel #14
           HANDLE: <>t3 #15
           LO: <>rel(read) #16
           EV: <>e1 #17
           SU: <>x1 ^7
           C1: <>x2 #18
        **** semrel_qua #19
           HANDLE: <>t4 #20
           LO: <>det(two) #21
```

```
           BV: <>x2 ^18
           RS: <>t5 #22
           SC: <>h2 #23
         **** semrel_pr #24
           HANDLE: <>t5 ^22
           LO: <>noun(book) #25
           IN: <>x2 ^18
      >--------
----- [consequently]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * adv #3
    HDTOKEN: <>t1 ^2
    MOD: * mod_type #4
      PRE: * sign #5
        HEAD: * v #6
          HDTOKEN: <>h1 #7
        MARKING: * unmarked #8
        CSTR: * leaf #9
      POST: * none #10
  SPR: []
  COMPS: []
  MARKING: * unmarked #11
  CONT: * cont #12
    LZT: ----- #13 ---<
        **** semrel_pr #14
           HANDLE: <>t1 ^2
           LO: <>adv(consequently) #15
           IN: <>ltop(h1) #16
      >--------
    WTOP: <>t1 ^2
```

```
    HCONS: <>[] #17
    VARUSE: <>[handle(h1)] #18
----- [two,students,intentionally,erased,three,
       files]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>det(two) #6
           BV: <>x1 #7
           RS: <>t2 #8
           SC: <>h1 #9
        **** semrel_pr #10
           HANDLE: <>t2 ^8
           LO: <>noun(student) #11
           IN: <>x1 ^7
        **** semrel_rel #12
           HANDLE: <>t3 #13
           LO: <>adv(intentionally) #14
           EV: <>e1 #15
           SU: <>x1 ^7
           C1: <>ltop(t4) #16
        **** semrel_rel #17
           HANDLE: <>t4 #18
           LO: <>rel(erased) #19
           EV: <>e1 ^15
           SU: <>x1 ^7
           C1: <>x2 #20
        **** semrel_qua #21
           HANDLE: <>t5 #22
```

```
           LO: <>det(three) #23
           BV: <>x2 ^20
           RS: <>t6 #24
           SC: <>h2 #25
         **** semrel_pr #26
           HANDLE: <>t6 ^24
           LO: <>noun(file) #27
           IN: <>x2 ^20
      >--------
----- [intentionally]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * adv #3
    HDTOKEN: <>t1 ^2
    MOD: * mod_type #4
      PRE: * sign #5
        HEAD: * v #6
          HDTOKEN: <>_13205 #7
          INTER: * semrel_rel #8
            EV: <>e1 #9
            SU: <>x1 #10
        MARKING: * unmarked #11
        CSTR: * leaf #12
      POST: * none #13
  SPR: []
  COMPS: []
  MARKING: * unmarked #14
  CONT: * cont #15
    LZT: ----- #16 ---<
        **** semrel_rel #17
           HANDLE: <>t1 ^2
```

```
          LO: <>adv(intentionally) #18
          EV: <>e1 ^9
          SU: <>x1 ^10
          C1: <>ltop(_13205) #19
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #20
    VARUSE: <>[thing(x1),event(e1)] #21
----- [two,students,probably,erased,three,files]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>det(two) #6
           BV: <>x1 #7
           RS: <>t2 #8
           SC: <>h1 #9
        **** semrel_pr #10
           HANDLE: <>t2 ^8
           LO: <>noun(student) #11
           IN: <>x1 ^7
        **** semrel_pr #12
           HANDLE: <>t3 #13
           LO: <>adv(probably) #14
           IN: <>ltop(t4) #15
        **** semrel_rel #16
           HANDLE: <>t4 #17
           LO: <>rel(erased) #18
           EV: <>e1 #19
           SU: <>x1 ^7
```

```
            C1: <>x2 #20
         **** semrel_qua #21
            HANDLE: <>t5 #22
            LO: <>det(three) #23
            BV: <>x2 ^20
            RS: <>t6 #24
            SC: <>h2 #25
         **** semrel_pr #26
            HANDLE: <>t6 ^24
            LO: <>noun(file) #27
            IN: <>x2 ^20
       >--------
----- [galileo,seeks,a,unicorn]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
         **** semrel_qua #4
            HANDLE: <>t1 #5
            LO: <>name(Galileo) #6
            BV: <>x1 #7
            RS: <>nil #8
            SC: <>h1 #9
         **** semrel_rel #10
            HANDLE: <>t2 #11
            LO: <>rel(seeks) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t4) #14
         **** semrel_qua #15
            HANDLE: <>t3 #16
            LO: <>det(a) #17
```

```
          BV: <>x2 #18
          RS: <>t4 #19
          SC: <>h2 #20
        **** semrel_pr #21
          HANDLE: <>t4 ^19
          LO: <>noun(unicorn) #22
          IN: <>x2 ^18
      >--------
----- [seeks]
----- 1 parse(s). all selection:
* sign #1
  TOKEN: <>t1 #2
  HEAD: * v #3
    HDTOKEN: <>t1 ^2
    INTER: * semrel_rel #4
      EV: <>e1 #5
      SU: <>x1 #6
    MOD: * mod_type #7
      PRE: * none #8
      POST: * none #9
    VFORM: * present #10
  SPR: ----- #11 ---<
      **** sign #12
         HEAD: * nom #13
           INTER: * semrel_qua #14
             BV: <>x1 ^6
           NUM: * sing #15
           CASE: * subc #16
           PERS: * p3 #17
         SPR: []
         COMPS: []
    >--------
```

```
  COMPS: ----- #18 ---<
      **** sign #19
         HEAD: * nom #20
           HDTOKEN: <>h1 #21
           CASE: * objc #22
         SPR: []
         COMPS: []
    >--------
  MARKING: * unmarked #23
  CONT: * cont #24
    LZT: ----- #25 ---<
        **** semrel_rel #26
           HANDLE: <>t1 ^2
           LO: <>rel(seeks) #27
           EV: <>e1 ^5
           SU: <>x1 ^6
           C1: <>ltop(h1) #28
      >--------
    WTOP: <>t1 ^2
    HCONS: <>[] #29
    VARUSE: <>[handle(h1),thing(x1),event(e1)] #30
----- [galileo,tries,to,find,a,unicorn]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>name(Galileo) #6
           BV: <>x1 #7
           RS: <>nil #8
           SC: <>h1 #9
```

```
        **** semrel_rel #10
           HANDLE: <>t2 #11
           LO: <>rel(tries) #12
           EV: <>e1 #13
           SU: <>x1 ^7
           C1: <>ltop(t4) #14
        **** semrel_rel #15
           HANDLE: <>t4 #16
           LO: <>rel(find) #17
           EV: <>e2 #18
           SU: <>x1 ^7
           C1: <>x2 #19
        **** semrel_qua #20
           HANDLE: <>t5 #21
           LO: <>det(a) #22
           BV: <>x2 ^19
           RS: <>t6 #23
           SC: <>h3 #24
        **** semrel_pr #25
           HANDLE: <>t6 ^23
           LO: <>noun(unicorn) #26
           IN: <>x2 ^19
      >--------
----- [mary,believes,that,tully,denounced,catiline]
----- 1 parse(s). lzt selection:
* sign #1
  CONT: * cont #2
    LZT: ----- #3 ---<
        **** semrel_qua #4
           HANDLE: <>t1 #5
           LO: <>name(Mary) #6
           BV: <>x1 #7
```

```
            RS: <>nil #8
            SC: <>h1 #9
         **** semrel_rel #10
            HANDLE: <>t2 #11
            LO: <>rel(believes) #12
            EV: <>e1 #13
            SU: <>x1 ^7
            C1: <>ltop(t5) #14
         **** semrel_qua #15
            HANDLE: <>t4 #16
            LO: <>name(Tully) #17
            BV: <>x2 #18
            RS: <>nil ^8
            SC: <>h2 #19
         **** semrel_rel #20
            HANDLE: <>t5 #21
            LO: <>rel(denounced) #22
            EV: <>e2 #23
            SU: <>x2 ^18
            C1: <>x3 #24
         **** semrel_qua #25
            HANDLE: <>t6 #26
            LO: <>name(Catiline) #27
            BV: <>x3 ^24
            RS: <>nil ^8
            SC: <>h3 #28
      >--------
{/modus/export3/staff/matsd/cs/code_toksem/
sentences.pl consulted, 2780 msec 97168 bytes}

yes
| ?-
```

5 How to download and run the implementation

The URL '`http://stp.ling.uu.se/~matsd/code/petfsg/`' contains the PETFSG file `petfsg.pl` and the other necessary files are found in the URL '`http://stp.ling.uu.se/~matsd/code/tbmrs/`'. The system requires SICStus Prolog (and Tcl/TK if the graphics is used).

A convenient way of starting a session with PETFSG and the present implementation is to `consult` a file with the following content (found in the file `load.pl`):

```
:-consult('petfsg.pl').
:-consult('tbmrspro.pl').
:-prepare_compilation.
:-compile_grammar('tbmrs.pl').
```

Here, the file `petfsg.pl` is assumed to contain the PETFSG system. The file `tbmrspro.pl` contains definitions of the predicates used by the grammar, and `tbmrs.pl` is the TBMRS grammar documented here. See the previous paper for details on PETFSG. The example analyses above have been generated from the file `examples.pl`, which contains the required `pp/2` calls. When `examples.pl` is consulted the example analyses are generated.

References

Dahllöf, M.: 1999a, 'Prolog-embedding feature structure grammar', *RUUL* (Reports from Uppsala University Department of Linguistics) **34** (this volume), 3–37.

Dahllöf, M.: 1999b, 'Token-Based Minimal Recursion Semantics: Integrating Davidson's Paratactic Treatment of Intensional Constructions in a Formal Syntax and Semantics'. Unpublished manuscript, available from the author.

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

RUUL (Reports from Uppsala University Dept of Linguistics) **34**, 101–123.

Predicate-functor logic

Mats Dahllöf
Department of Linguistics, Uppsala University
E-mail: `Mats.Dahllof@ling.uu.se`
September, 1999

1 Introduction

This paper is about the semantic work done by variables and variable-binding operators in predicate calculus and similar formalisms. Quine has shown that variables can be replaced, without loss or gain in semantic power, by other formal devices. In this way we get a new "picture" of how predicate calculus works. This investigation is motivated, on the one hand, by the fact that variables are very important, and on the other hand by the conceptually problematic "semantics" of variables: They serve a semantic purpose, but do not have an ordinary semantic value.

One way of investigating what a device is good for is to remove it and see what other instruments would be needed to do its work. Variables will be investigated in this way here.

2 Ordinary predicate calculus with variables

Only atomic expressions denote predicates in standard predicate calculus: It lacks a direct compositional semantics for

predicates and deserves the name "*predicate* calculus" only indirectly: A formula with n different free variables does, in a sense, stand for an n-place predicate, but not simply by denoting one. Compositionally derived predicates are "present" as formulae which are true (or false) relative to variable-assignment functions. Variables necessitate a notion of truth (of formulae) as relative not only to an interpretation, but also to a variable-assigment function. The notion of variable-assignment-relative truth is necessary as a means of defining the main (only *interpretation*-relative) notion of truth.

From the point of view of linguistics, interpretation-relative denotations like truth values and predicates (extensions) have a clear connection to intuitive semantic properties, whereas interpretation-and-variable-assignment-relative truth is a much more exotic notion, at least as used in predicate calculus. A simple statement like "someone sleeps" would, for instance, be rendered as "$\exists x S(x)$". The introduction of variables here (and in other semantic constructions) shows that predicate calculus is very different from natural language with respect to its modes of composition. (This use of variables and variable-binding operators is due to Frege's genial and ingenious logic *Begriffsschrift*.)

If we try to view open formulae as predicate-denoting expressions, variables complicate the picture considerably: Formula (1), for instance, may be intended to stand for the three-place relation that holds of three objects when the first (x) is larger than the second (y) and the second is larger than the third (z).

$$(1) \qquad \textsf{larger-than}(x, y) \wedge \textsf{larger-than}(y, z)$$

The two subordinate formulae "$\textsf{larger-than}(x, y)$" and "$\textsf{larger-than}(y, z)$" intuitively stand for the same two-place relation. The important thing is that two different variables

occupy the argument places (otherwise we would have defined the property "larger than itself"). The choice of variables is immaterial from the point of view of the subformulae themselves. When they are conjoined, however, the fact that the second argument position of the first subformula is occupied by a variable identical to that found in the first argument position of the second subformula is crucial, as the two argument positions thereby are connected in a semantically significant way. The conjunction of two instances of a two-place predicate in this way defines a three-place predicate. If we take the semantic value (relative to an interpretation) of a formula with free variables to be the predicate it in this way defines (given an ordering of the variables), then we find that predicate logic with variables does not possess a compositional semantics:[1] The semantic value of the conjunction in (1) is not possible to determine from the semantic values of the conjuncts, because of the significance of the identity of variables. A formula like (2), below, would define another predicate, but is just like formula (1) a conjunction whose conjuncts (we have supposed)

[1]It might be objected here that variables are only a syntactic device and that their only function is to define by which "mode of combination" predicates are joined together. This suggestion implies that there is an infinity of such modes of combination, as there is no upper limit to the number of variables that may occur in a formula. (Compositionality is thereby in a sense saved: The semantic value of the whole is determined by the semantic values of its constituents and their modes of combination.) However, having an infinite number of modes of combination is clearly unacceptable and this view of the the syntax is inadequate: Variable cooccurrence is not a matter of syntactic rule application. For instance, the conjunction rule in an ordinary first-order logic syntax like the one given by Dowty, Wall, and Peters (1981: 56–57) (rule B. 3) or by Allwood, Andersson, and Dahl (1977: 71–72) (rule (g) (iii)) just says that two formulae joined by a conjunction sign form a new formula. The syntactic combination rules do not mention the variables occurring in the formulae.

stand for the relation of being larger than.

(2) $\textsf{larger-than}(x, y) \wedge \textsf{larger-than}(x, z)$

This lack of "naive" compositionality is, of course, due to the presence of variables. There are *two* systems of semantically significant structure in ordinary predicate calculus: the constituent structure (as defined by the syntactic modes of combination) and what we could call the variable cooccurrence structure.

A variable does not contribute to the semantic value of an expression by having a certain semantic value itself. Its significance is due to its being bound from the outside, by being identical to another variable token occurring in another expression. So, when we combine the two conjuncts in formula (2) into a conjunction, the semantic value of the conjunction depends on the semantic values of the conjuncts and on *both* the semantic operation represented by conjunction and the identification of two variable occurrences. The latter factor is reflected neither in the syntactic operation (conjoining) or in the semantic values of the conjuncts (as naively understood). The difficult situation lies in the fact that semantically significant links within a formula are established both by means of constituent syntax (as defined by the syntactic rules) and by means of variable cooccurrence.

A traditional semantics for ordinary predicate calculus deals with this situation and establishes compositionality with the help of certain technical manœuvres, which invalidate the idea that open formulae represent predicates. One method is to define semantic values as relative to an interpretation *and* a *value assignment*, which is a function assigning values to variables.[2]

[2]See Dowty, Wall, and Peters (1981: 59–61) for a semantics for first-order predicate calculus along these lines.

(Other methods have to introduce some similar kind of further relativity.) Open formulae are then taken to denote a truth value relative to an interpretation *and* a value assignment function. The truth value of formulae without free variables will not be sensitive to the value assignment, and *their* truth values are consequently only interpretation-dependent.

This method saves variables and compositionality. The disadvantage is that the system of semantic values becomes more complex and artificial: Instead of having the intuitively simple picture with only true expressions (sentences), and expressions true of extra-logical entities (predicates), we also have to accept expressions true relative to variable assignments. This complication also comes with the *restriction* that predicate expressions cannot be compositional. (Addition of λ-terms may be a way of removing that restriction in a variable-based calculus.)

The problems may be summarized as follows: The only kind of interpretation-relative complex denotations are associated with predicate constants, which are atomic symbols, and the syntactically complex expressions (formulae) are only assigned truth-values, which do not possess any significant structure, by interpretations. *Syntactically simple objects denote complex ones, and syntactically complex objects denote simple ones*, to put it briefly.

Variables (and the difficulties they introduce) are in no way essential to first-order predicate calculus (or to most other variable-based calculi). Quine has shown this by inventing a variable-free predicate calculus. His calculus uses a small number of functors, whereby new predicates are defined in terms of primitive or already defined one. These functors are responsible for cross-identification and rearrangement of argument places, and quantification. They take over the tasks performed by variables. This *predicate-functor logic* is precisely as powerful

as ordinary first-order predicate calculus (with variables and variable-binding quantifiers). This variable-free language is semantically perspicuous: Every expression denotes a semantic value of the intuitively right kind (i.e. a truth value, a predicate extension, or an individual) relative to an interpretation and the semantics is perfectly compositional. Quine's calculus consequently offers an alternative way of organizing the compositional semantics of first-order predicate calculus (viewed as a category of calculus defined by its semantic power).

3 Predicate-functor logic

The most important aspect of a predicate-functor logic is its inventory of predicate-functors. Quine (1960, 1971, 1976a, 1981a, 1981b, also cf. 1976b) has suggested a few alternative selections. Between four and seven predicate-functors are required. Quine's languages do not contain individual constants, but the calculus presented below will.[3]

I will use Quine's set of six functors from the 1960 paper 'Variables Explained Away'. The notation will however be slightly modified to suit my purposes (and taste) better. Quine (1976a) has shown how to manage with only four functors, but this gain in economy leads to a corresponding complication of expression. I think that the 1960 set is easier to understand and use. The language discussed here is a first-order logic with identity (i.e. an identity predicate is included among the logi-

[3]For some results on predicate-functor logic, see Noah (1980), Grünberg (1983), Kuhn (1983), and Bacon (1985). Predicate-functor logic is applied to natural language semantics by Grandy (1976), who suggests that "linguists, psychologists and philosophers should pay more attention to algebraic formulations of logic when discussing the logical form of natural language sentences" [p. 398], and by Purdy (1991).

cal constants). There are no other logical constants than these seven (viz. the six functors and the identity predicate).[4]

There are two kinds of terms in the present predicate-functor logic: *predicate terms* and *individual constants*. A predicate term consists of a predicate constant or is a complex expression. Predicate terms are of a certain degree (arity). Formulae may be identified with predicate terms of degree 0. The six predicate functors will here be represented by the symbols "$\imath$", "$\bar{\imath}$", "Σ", "$\mathfrak{C}$", "$\boxtimes$", and "$\beth$". I will use a predicate-first notation, in order to eliminate the need for parentheses. This means that a predicate will precede its individual constant arguments, but that predicate functors will be placed after the predicate term(s) they operate on. (This notation makes it natural to think of individual constants as being applied to predicate terms rather than vice versa.) This syntax implies that complex predicate terms are always formed by strings whose first element is a predicate constant and whose last element is a functor or an individual constant (as complex predicate terms are formed by the application of a functor or an individual constant.) Reading from left to right gives us a bottom-up perspective on how predicates given by the predicate constants are modified or combined into new ones by means of functor applications.

The two functors "$\imath$" and "$\bar{\imath}$" are used to permute argument places in predicates. Applied to a predicate term they yield a predicate term of the same degree. The *minor inversion* functor "$\imath$" interchanges the two first argument places, whereas the *major inversion* functor "$\bar{\imath}$" puts the first argument place last, so to speak. (Applied to a two-place predicate the major and minor inversions are identical.) These functors are necessary because

[4]Those who are interested in the details of Quine's languages and want to compare them to each other and to the one defined here are referred to his papers.

the "Σ" and "$\beth$" functors (see below) operate on the first one or two argument places of a predicate. It is therefore crucial that any argument place can be "moved" to the position where a "Σ" or "$\beth$" functor can "reach" it. If we use a kind of mixture notation we may describe the effects of the two inversion functors as follows (taking variables to be implicitly universally quantified and P to be an arbitrary n-place predicate).

$$P\imath x_2 x_1 x_3 \ldots x_n \Leftrightarrow P x_1 x_2 x_3 \ldots x_n$$
$$P\bar{\imath} x_2 \ldots x_n x_1 \Leftrightarrow P x_1 x_2 \ldots x_n$$

The two kinds of inversion are sufficient to define any argument place permutation. (There are $n!$ such "permutations" of an n-place predicate, including its original argument place ordering.) For instance, the fact that all six argument place permutations of a three-place predicate may be obtained by applications of these two operators is illustrated by the following six equivalent formulae (note that "$((((P\imath)\bar{\imath})a)c)b$" is the syntactic structure of "$P\imath\bar{\imath}acb$"):

$$Pabc \Leftrightarrow P\imath\bar{\imath}acb \Leftrightarrow P\imath bac \Leftrightarrow P\bar{\imath}bca \Leftrightarrow P\overline{\imath\imath}cab \Leftrightarrow P\bar{\imath}\imath cba$$

It is easy to verify that the six formulae above are equivalent. Note also that further applications of the "$\imath$" and "$\bar{\imath}$" functors on the predicate terms can only yield a new predicate term whose denotation is identical to the denotation of one of the formulae above. So, if $[\![x]\!]$ is the semantic value (i.e. denotation) of an expression x (relative to a given interpretation), $[\![P\imath\imath]\!] = [\![P\overline{\imath\imath}\imath]\!] = [\![P]\!]$ (provided P is of degree 3). And, if P is a two-place predicate, we have, for instance, $[\![P]\!] = [\![P\imath\imath]\!] = [\![P\bar{\imath}\imath]\!] = [\![P\imath\bar{\imath}]\!]$ and $[\![P\imath]\!] = [\![P\imath\imath\imath]\!] = [\![P\bar{\imath}\imath\bar{\imath}]\!] = [\![P\imath\overline{\imath\imath}]\!]$.

The *reflection* functor "Σ" is used to identify two argument places (compare a reflexive pronoun). It operates on an at least two-place predicate term, identifies the first two argument

places, and consequently yields a predicate term whose degree is one step lower than the term to which it is applied. If "L" is a two-place predicate symbol corresponding to the verb *love*, "LΣ" corresponds to *love oneself*. Generally it holds:

$$P\Sigma x_1 \ldots x_n \Leftrightarrow Px_1x_1 \ldots x_n$$

This functor ("Σ") encodes information that is encoded by means of variables in ordinary predicate calculus, where the identification of argument places is achieved by associating them with the same variable. One of the reasons that the inversion functors ("$\imath$" and "$\bar{\imath}$") are of crucial importance is that they allow any two argument places to be "moved" to the front positions where the "Σ" functor may be used to unite them.

The *complement* functor[5] applies to a predicate term and yields the predicate that holds in all those cases where the first predicate does not hold. So, if "L" is as above, "L$\mathfrak{C}$" stands for the relation of not loving. Applied to a formula, the complement functor yields one with the reverse truth value. The term obtained by an application of the complement functor is of the same degree as the operand term. The following formula (which again is expressed in mixture notation) clarifies the semantics of "$\mathfrak{C}$":

$$P\mathfrak{C}x_1 \ldots x_n \Leftrightarrow \neg(Px_1 \ldots x_n)$$

The *Cartesian multiplication* functor "$\boxtimes$" is the only one that operates on two predicate terms. The degree of the Cartesian product is equal to the sum of the degrees of the operand predicate terms. If the operands represent the predicates P and Q, whose degrees are m and n respectively, their Cartesian product is the relation that holds of $m+n$ ordered individuals if and

[5]It is called "negation" in Quine (1960), but the term "complement" is used in the later papers.

only if P holds of the first m ones and Q holds of the remaining n ones. This is made clearer by the following equivalence:

$$PQ\boxtimes x_1 \ldots x_m y_1 \ldots y_n \Leftrightarrow (Px_1 \ldots x_m \wedge Qy_1 \ldots y_n)$$

So, if "L" is as above, and "H" stands for the predicate of being happy, "HL⊠" stands for the three-place relation holding of three individuals if and only if the first is happy and the second loves the third. An application of "⊠" does not connect any argument places. (To do that is the purpose of the "Σ" functor.) No connection between the three argument places in "HL⊠" example is consequently implied. By "concatenating" argument sequences in this way, the Cartesian multiplication functor makes it possible to apply the other functors in ways that connect argument places. For instance, "HL⊠Σ" stands for the relation that holds of two individuals if and only if the first is happy and loves the second.

The *cropping* or *derelativization* functor "⊐", finally, "removes" the first argument place by introducing an existential quantification. It consequently yields a predicate term whose degree is one step lower than that of the operand term. In mixture notation we may characterize its semantics in this way:

$$P\sqsupset x_2 \ldots x_n \Leftrightarrow \exists x_1[Px_1 \ldots x_n]$$

If "L" is as before, "L⊐" stands for the predicate of being loved by someone (the "subject" being the first argument of "L"), and "L⊐⊐" for the formula saying that someone loves someone. The "⊐" functor only operates on the first argument place of the predicate represented by the predicate term to which it applies. The importance of the inversion functors ("$\imath$" and "$\bar{\imath}$") in this calculus is again illustrated: They can "move" an argument place to the position where the "⊐" functor may excise it (by cutting off the first item on each n-tuple in the predicate extension).

The predicate-functor logic based upon the six functors introduced here is as powerful as ordinary predicate logic (cf. Quine's papers on the subject). I will not produce a formal proof of this claim, but the discussion here and the examples in Section 3.3 should make this assertion convincing. A formal and quite detailed definition of the syntax and semantics of the present predicate-functor logic will now be given. We will then return to the question of how this notation relates to that of ordinary predicate calculus with variables. A few examples will illustrate this.

3.1 A formal summary of the syntax

Let us call the present formalism *Predicate-Functor Logic with Individual Constants*, abbreviated to PFLIC. The *lexicon* of a logical formalism contains the non-logical constants of the language. In the case of PFLIC, it contains two kinds of constants: individual constants and predicate constants. A predicate constant is of a certain degree (arity) n and $n \geq 0$. (A PFLIC lexicon is just like the one an ordinary predicate calculus is based on.)

The following principles define the syntax of an instance of PFLIC if a lexicon is given. There are two kinds of expressions in an instance of PFLIC, *individual constants* and *predicate terms*. A predicate term is of a certain degree n and $n \geq 0$. Predicate terms of degree 0 are formulae (definition).

In the formulation of the syntax and semantics, I will use "P" and "Q" as meta-variables standing for arbitrary predicate terms, "i" to stand for an arbitrary individual constant, and "Φ" and "Ψ" to stand for arbitrary formulae. The six predicate functor symbols and the identity predicate symbol will be used in the meta-language to denote the typographi-

cally identical symbols of PFLIC. A pair or triple of juxtaposed meta-variables stands for the expression obtained by concatenating, in the given order, the expressions denoted by the two or three meta-variables.

Individual constants are given directly by the lexicon, but predicate terms may be syntactically complex. The following rules define their syntax.

- A predicate constant of degree n is a predicate term of degree n.
- The identity symbol $\mathbb{I}$ is a predicate term of degree 2.
- If P is a predicate term of degree n, where $n \geq 1$, and i is a individual constant, then Pi is a predicate term of degree $n - 1$.
- If P is a predicate term of degree n, where $n \geq 2$, then $P\imath$ (the minor inversion of P) is a predicate term of degree n.
- If P is a predicate term of degree n, where $n \geq 2$, then $P\bar{\imath}$ (the major inversion of P) is a predicate term of degree n.
- If P is a predicate term of degree n, where $n \geq 2$, then $P\Sigma$ (the reflection of P) is a predicate term of degree $n - 1$.
- If P is a predicate term of degree n, then $P\mathfrak{C}$ (the complement of P) is a predicate term of degree n.
- If P is a predicate term of degree m, and Q is a predicate term of degree n, then $PQ\boxtimes$ (the Cartesian product of P and Q) is a predicate term of degree $m + n$.
- If P is a predicate term of degree n, where $n \geq 1$, then $P\rceil$ (the derelativization of P) is a predicate term of degree $n - 1$.

There are no predicate terms except those defined by these principles.

3.2 A formal semantics

The semantics of PFLIC is to be defined with the help of ordinary model-theoretic techniques. An interpretation of an instance of PFLIC assigns semantic values to its non-logical constants and the semantic rules then fixes the semantic values of the composite expressions (all of which are predicate terms). An interpretation is an ordered pair $\langle \mathcal{D}, \mathcal{F} \rangle$, where $\mathcal{D}$ is a domain, i.e. the set of entities over which quantification ranges, and $\mathcal{F}$ an assignment function assigning a semantic value to each item in the lexicon. The two truth values are TRUE and FALSE. The following restrictions hold with respect to any interpretation $\mathcal{M} = \langle \mathcal{D}, \mathcal{F} \rangle$:

- If i is an individual constant, then $\mathcal{F}(i) \in \mathcal{D}$.
- If P is a predicate constant of of degree n, where $n \geq 1$, then $\mathcal{F}(P) \subseteq \mathcal{D}^n$.
- If P is a predicate constant of of degree 0, then $\mathcal{F}(P) \in \{\text{TRUE}, \text{FALSE}\}$.

The following equations define the semantic values of composite predicate terms (given an interpretation $\mathcal{M} = \langle \mathcal{D}, \mathcal{F} \rangle$). Cases involving formulae are treated separately from those involving other predicate terms.

- If c is a non-logical constant, $[\![c]\!] = \mathcal{F}(c)$.
- $[\![\mathbb{I}]\!] = \{\langle x, x \rangle \mid x \in \mathcal{D}\}$.

- If Pi is a predicate term of degree n, where $n \geq 1$ and P is a predicate term of degree $n+1$ and i a individual constant, then $[\![Pi]\!] = \{\langle x_1, \ldots, x_n\rangle \mid \langle [\![i]\!], x_1, \ldots, x_n\rangle \in [\![P]\!]\}$.

- If $P\imath$ is a predicate term, where P is a predicate term of degree n, $n \geq 2$, then
  $[\![P\imath]\!] = \{\langle x_2, x_1 \ldots, x_n\rangle \mid \langle x_1, x_2, \ldots, x_n\rangle \in [\![P]\!]\}$.

- If $P\bar{\imath}$ is a predicate term, where P is a predicate term of degree n, $n \geq 2$, then
  $[\![P\bar{\imath}]\!] = \{\langle x_2, \ldots, x_n, x_1\rangle \mid \langle x_1, x_2, \ldots, x_n\rangle \in [\![P]\!]\}$.

- If $P\Sigma$ is a predicate term, where P is a predicate term of degree n, $n \geq 2$, then
  $[\![P\Sigma]\!] = \{\langle x_1, \ldots, x_n\rangle \mid \langle x_1, x_1, \ldots, x_n\rangle \in [\![P]\!]\}$.

- If $P\mathfrak{C}$ is a predicate term, where P is a predicate term of degree n, where $n \geq 1$, then $[\![P\mathfrak{C}]\!] = \mathcal{D}^n - [\![P]\!]$.

- If $PQ\boxtimes$ is a predicate term of degree $m + n$, where P is a predicate term of degree m, $m \geq 1$ and Q is a predicate term of degree n, $n \geq 1$, then $[\![PQ\boxtimes]\!] = \{\langle x_1, \ldots, x_m, y_1, \ldots, y_n\rangle \mid \langle x_1, \ldots, x_m\rangle \in [\![P]\!]$ and $\langle y_1, \ldots, y_n\rangle \in [\![Q]\!]\}$.

- If $PQ\boxtimes$ is a predicate term of degree n, where P is a formula (i.e. a predicate term of degree 0), and Q is a predicate term of degree n, $n \geq 1$, then, if $[\![P]\!] =$ TRUE, $[\![PQ\boxtimes]\!] = [\![Q]\!]$, otherwise (i.e. if $[\![P]\!] =$ FALSE), $[\![PQ\boxtimes]\!] = \emptyset$.

- If $PQ\boxtimes$ is a predicate term of degree n, where P is a predicate term of degree n, $n \geq 1$, and Q is a formula (i.e. a predicate term of degree 0), then, if $[\![Q]\!] =$ TRUE, $[\![PQ\boxtimes]\!] = [\![P]\!]$, otherwise (i.e. if $[\![Q]\!] =$ FALSE), $[\![PQ\boxtimes]\!] = \emptyset$.

- If $P\beth$ is a predicate term, where P is a predicate term of degree n, $n \geq 2$, then $[\![P\beth]\!] = \{\langle x_2, \ldots, x_n\rangle \mid$ there is an x_1 such that $\langle x_1, x_2, \ldots, x_n\rangle \in [\![P]\!]\}$.

- If Pi is a formula, where P is a predicate term of degree 1 and i an individual constant, then $[\![Pi]\!] =$ TRUE, if $[\![i]\!] \in [\![P]\!]$, otherwise $[\![Pi]\!] =$ FALSE.

- If $\Phi\mathfrak{C}$ is a formula, then $[\![\Phi\mathfrak{C}]\!] =$ TRUE if $[\![\Phi]\!] =$ FALSE, otherwise $[\![\Phi\mathfrak{C}]\!] =$ FALSE.

- If $\Phi\Psi\boxtimes$ is a formula, then $[\![\Phi\Psi\boxtimes]\!] =$ TRUE if $[\![\Phi]\!] =$ TRUE and $[\![\Psi]\!] =$ TRUE, otherwise $[\![\Phi\Psi\boxtimes]\!] =$ FALSE.

- If $P\beth$ is a formula (where P is a predicate term of degree 1), then $[\![P\beth]\!] =$ TRUE if $[\![P]\!] \neq \emptyset$, otherwise $[\![P\beth]\!] =$ FALSE.

3.3 How the functors work

Ordinary predicate calculus with variables and PFLIC are semantically equivalent in this sense: If they share a lexicon (of non-logical constants), every PFLIC formula corresponds to an "ordinary" formula with precisely the same truth-conditions, and *vice versa*. (Interpretations may be shared by an instance of PFLIC and one of ordinary first-order predicate calculus if the two languages contain the same non-logical constants.) Some discussion and examples should make the truth of this claim easier to see and facilitate the understanding of the PFLIC semantics.

A minor difference in notation is that in PFLIC predicate terms combine with one individual constant argument into a predicate term of a one step lower degree. So, "$P(a, b)$" in ordinary notation corresponds to "Pab" in the notation introduced here, "P" being two-place. In this formula, "Pa" is a

subexpression denoting a one-place predicate (that of being something which a is related to by the P-relation). In the ordinary predicate logic notation no constituent corresponds to this subexpression. (There are no composite expressions corresponding to predicates at all, as mentioned previously)

Two functors in PFLIC do the work of the ordinary truth-functional operators. They are the complement functor ("𝔈") and the Cartesian multiplication functor ("⊠"), which correspond to negation and conjunction, respectively, when applied to formulae. These operators do not only operate on formulae, but also on predicate terms (which denote predicates). This means that we can conjoin predicate terms without having to supply the arguments that would combine with them into formulae.[6] So, if we assume that both "P" and "Q" represent two-place predicates and that "a", "b", "c", and "d" are individual constants, the formulae "$PabQcd⊠$" and "$PQ⊠abcd$" are equivalent. Any truth conditional composition of formulae can be defined in terms of negation and conjunction and the absence of disjunction, material implication and other truth-functional operators consequently does not weaken the expressive power of PFLIC.

The derelativization operator ("ℶ") represents an existential quantification applying to the first argument place of a predicate. Universal quantification is easily defined in terms of the complement functor and existential quantification. (Remem-

[6]The Cartesian multiplication operator is in this respect closer to natural language conjunction than is the ordinary purely truth-functional conjunction operator found in ordinary propositional and predicate calculus. However, natural language conjunction of (say) two property adjectives does not produce a two-place predicate expression. So, "ugly and expensive" corresponds to "UE⊠Σ" (the property of being both ugly and expensive), rather than simply to "UE⊠" (the relation that holds between any ugly thing and any expensive thing).

ber that generally $\forall x\Phi(x) \Leftrightarrow \neg\exists x\neg\Phi(x)$.) The quantification expressed by the "$\boxed{\ }$" operator only involves the first argument place. This means that we must be able to connect argument places. This is done by means of the reflection functor "Σ", which connects the first two argument places of a predicate. The formula "$P\Sigma\boxed{\ }$", for instance, corresponds to "$\exists xP(x,x)$" in ordinary syntax, where the identification of the two argument places is expressed by associating the same variable with the two places. In PFLIC this identification is performed by means of an application of "Σ".

Argument places must often be "moved" to come within the reach of the Σ functor. This need motivates the inversion functors "$\imath$" and "$\overline{\imath}$". They allow us to obtain a predicate term standing for an arbitrarily permuted version of any predicate defined by a predicate term. In particular, any two argument places that we would like to identify by means of "Σ" can be "moved" to the two initial positions.

If we consider formulae in a notation mixing elements from ordinary predicate calculus and PFLIC in a way that allows PFLIC predicate terms to occupy the position of ordinary predicate calculus predicate symbols, we can show the stepwise process of conversion from the one notation to the other. (The semantics of this mixture calculus is intuitively obvious and it would not be difficult to produce a formal semantics for it.) It should be noted that the order in which these steps are exhibited here is immaterial. Let us see how "$\forall x[\mathsf{M}(x) \rightarrow \mathsf{L}(x)]$" is recast to conform to the PFLIC syntax. (An expression within a box is a PFLIC predicate term.)

$\forall x[\boxed{\mathsf{M}}(x) \rightarrow \boxed{\mathsf{L}}(x)]$	[formula in ordinary notation]
$\neg\exists x\neg[\boxed{\mathsf{M}}(x) \rightarrow \boxed{\mathsf{L}}(x)]$	[universal quant. eliminated]
$\neg\exists x\neg\neg[\boxed{\mathsf{M}}(x) \wedge \neg\boxed{\mathsf{L}}(x)]$	[implication eliminated]
$\neg\exists x[\boxed{\mathsf{M}}(x) \wedge \neg\boxed{\mathsf{L}}(x)]$	[double negation eliminated]
$\neg\exists x[\boxed{\mathsf{M}}(x) \wedge \boxed{\mathsf{L}\mathfrak{C}}(x)]$	[negation eliminated]
$\neg\exists x\boxed{\mathsf{ML}\mathfrak{C}\boxtimes}(x,x)$	[conjunction eliminated]
$\neg\exists x\boxed{\mathsf{ML}\mathfrak{C}\boxtimes\Sigma}(x)$	[argument places identified]
$\neg\boxed{\mathsf{ML}\mathfrak{C}\boxtimes\Sigma\beth}$	[existential quant. eliminated]
$\boxed{\mathsf{ML}\mathfrak{C}\boxtimes\Sigma\beth\mathfrak{C}}$	[negation eliminated]

If "M" and "L" stand for the predicates to be a man and to be lazy, respectively, "$\mathsf{ML}\mathfrak{C}\boxtimes\Sigma\beth\mathfrak{C}$" says that all men are lazy. Let us dissect this formula to examine its parts. The term "$\mathsf{L}\mathfrak{C}$" stands for the predicate of not being lazy, and "$\mathsf{ML}\mathfrak{C}\boxtimes$" for the relation that holds between any man and anyone who is not lazy. "$\mathsf{ML}\mathfrak{C}\boxtimes\Sigma$", then, stands for the property of standing in this relation to oneself, i.e. for the property of being a man without being lazy. "$\mathsf{ML}\mathfrak{C}\boxtimes\Sigma\beth$" is a formula saying that there is at least one individual who has this property. "$\mathsf{ML}\mathfrak{C}\boxtimes\Sigma\beth\mathfrak{C}$", finally, is the denial of this, i.e. it says that there is no individual who is a man without being lazy. In other words: all men are lazy.

This example illustrates the differences between the ordinary notation for first-order logic and the PFLIC notation. In the former, predicates are represented only by simple expressions. Predicate symbols combine with individual constants and variables into atomic formulae and these formulae are combined into more complex formulae by means of truth-functional composition, cross-identification of variables, and quantificational binding of variables. In PFLIC, on the other hand, everything is expressed by the modification or joining of predicate terms. In the mixture notation above, this difference is made clear

by the way the logical constants are successively incorporated into the predicate terms and the predicate terms conjoined into larger ones as we step by step transform a formula in ordinary notation into one in the PFLIC notation.

Let us consider another example. The logical form of one reading of a sentence like "Every logician has read something written by Frege" may be rendered as "$\forall x[\mathsf{L}(x) \rightarrow \exists y[\mathsf{W}(y, \mathsf{f}) \wedge \mathsf{R}(x, y)]]$" (constants being interpreted in the obvious fashion). We may convert it into PFLIC notation as follows:

$\forall x[\boxed{\mathsf{L}}(x) \rightarrow \exists y[\boxed{\mathsf{W}}(y, \mathsf{f}) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x\neg[\boxed{\mathsf{L}}(x) \rightarrow \exists y[\boxed{\mathsf{W}}(y, \mathsf{f}) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x\neg\neg[\boxed{\mathsf{L}}(x) \wedge \neg\exists y[\boxed{\mathsf{W}}(y, \mathsf{f}) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y[\boxed{\mathsf{W}}(y, \mathsf{f}) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y[\boxed{\mathsf{W}\imath}(\mathsf{f}, y) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y[\boxed{\mathsf{W}\imath\mathsf{f}}(y) \wedge \boxed{\mathsf{R}}(x, y)]]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y\boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes}(y, x, y)]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y\boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath}(x, y, y)]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y\boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}}(y, y, x)]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\exists y\boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma}(y, x)]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \neg\boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor}(x)]$
$\neg\exists x[\boxed{\mathsf{L}}(x) \wedge \boxed{\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor\mathfrak{C}}(x)]$
$\neg\exists x\boxed{\mathsf{L}\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor\mathfrak{C}\boxtimes}(x, x)$
$\neg\exists x\boxed{\mathsf{L}\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor\mathfrak{C}\boxtimes\Sigma}(x)$
$\neg\boxed{\mathsf{L}\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor\mathfrak{C}\boxtimes\Sigma\rfloor}$
$\boxed{\mathsf{L}\mathsf{W}\imath\mathsf{f}\mathsf{R}\boxtimes\imath\bar{\imath}\Sigma\rfloor\mathfrak{C}\boxtimes\Sigma\rfloor\mathfrak{C}}$

The constituent terms involved here may be interpreted as follows:

"$\mathsf{W}\imath\mathsf{f}$": the property of being written by Frege.

"W$\imath$fR$\boxtimes$": the relation that holds of three things iff[7] the first is written by Frege and the second has read the third.

"W$\imath$fR$\boxtimes\imath$": the relation that holds of three things iff the second is written by Frege and the first has read the third.

"W$\imath$fR$\boxtimes\imath\bar{\imath}$": the relation that holds of three things iff the first is written by Frege and the second has been read by the third.

"W$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma$": the relation that holds of two things iff the first is written by Frege and has been read by the second.

"W$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\beth$": the property of having read something written by Frege.

"W$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\beth\complement$": the property of not having read something written by Frege.

"LW$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\beth\complement\boxtimes$": the relation that holds between a logician and anyone who has not read something written by Frege.

"LW$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\beth\complement\boxtimes\Sigma$": the property of being a logician and not having read something written by Frege.

[7] The conjunction "iff" should be read as shorthand for "if and only if".

"LW$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\rceil\mathfrak{C}\boxtimes\Sigma\rceil$": the truth-value that there is a logician who has not read something written by Frege.

"LW$\imath$fR$\boxtimes\imath\bar{\imath}\Sigma\rceil\mathfrak{C}\boxtimes\Sigma\rceil\mathfrak{C}$": the truth-value that there is no logician who has not read something written by Frege.

These examples show how a formula in standard predicate calculus can be transformed into the PFLIC notation. An algorithm spelling out the general procedure in detail could be provided, but will not be specified here.

4 Concluding remarks

PFLIC is a semantically more parsimonious language than ordinary predicate calculus. The variable-relative aspect of the semantics is eliminated. As usual, the price we have to pay for a step from a semantically richer language to a more parsimonious one is a certain complication of expression in some cases. At least, PFLIC represents a "way of thinking" that probably feels quite awkward to people used to ordinary predicate calculus.

It is consequently obvious that the variable-based notation has some advantages. Variables give us a kind of graphical overview of logical structure. They indicate each "unfilled" argument position and show how they are connected (by being filled with the same variable). It may be quite difficult, by contrast, to keep track of the effects of the permutational inversion operators ("$\imath$" and "$\bar{\imath}$") in PFLIC. It is consequently not a good idea to try to convince the logic community to abandon variables in favour of predicate functors. However, variables do

complicate semantics, and in some contexts it may be a good idea to replace them by other devices (cf. Dahllöf 1995).

The main point of this paper has been to introduce the reader to predicate-functor logic and to convince him or her that it is a calculus of great theoretical interest. By seeing that and how predicate calculus can be given a "purer" but equally powerful semantics when variables are expelled, we elucidate what variables are good for. This is important, as the Fregean use of variables in the analysis of language has had a profound influence. Twentieth century linguistics and philosophy of language would certainly have looked quite different if some early modern logician had convinced the logic community that the predicate-functor-based way of dealing with quantification is the natural one.

References

Allwood, J., Andersson, L.-G., and Dahl, Ö., 1977, *Logic in Linguistics.* Cambridge, Cambridge University Press.

Bacon, J., 1985, 'The Completeness of Predicate-Functor Logic', *Journal of Symbolic Logic* **50**, 903–926.

Barwise, J. and Cooper, R. 1981, 'Generalized Quantifiers and Natural Language', *Linguistics and Philosophy* **4**, 159–219.

Dahllöf, M., 1995, *On the Semantics of Propositional Attitude Reports*, Göteborg University, Dept of Linguistics.

Dowty, D. R., Wall, R. E. and Peters, S., 1981, *Introduction to Montague Semantics*, Dordrecht, Reidel.

Grandy, R. E., 1976, 'Anadic Logic and English', *Synthese* **23**, 395–402.

Grünberg, T., 1983, 'A Tableau System of Proof for Predicate-Functor Logic with Identity', *Journal of Symbolic Logic* **48**, 1140–1144.

Kuhn, S. T., 1983, 'An Axiomatization of Predicate Functor Logic', *Notre Dame Journal of Formal Logic* **24**, 233–241.

Noah, A., 1980, 'Predicate-Functors and the Limits of Decidability in Logic', *Notre Dame Journal of Formal Logic* **21**, 701-707.

Purdy, W. C., 1991, 'A Logic for Natural Language', *Notre Dame Journal of Formal Logic* **32**, 409–425.

Quine, W. V., 1960, Variables Explained Away, *Proceedings of American Philosophical Society* **104**, 343–347. Also in Quine, W. V. (1960) *Selected Logic Papers*, New York, Random House, 227–235.

Quine, W. V., 1971, 'Predicate Functor Logic', in Fenstad, J. E. (ed.), *Proceedings of the Second Scandinavian Logic Symposium*, Amsterdam and London, North-Holland Publishing Company, 309–315.

Quine, W. V., 1976a, 'Algebraic Logic and Predicate Functors', in Quine, W. V., *The Ways of Paradox* (Revised and enlarged edition), Cambridge, Mass., Harvard University Press, 283–307.

Quine, W. V., 1976b, 'The Variable', in Quine, W. V., *The Ways of Paradox* (Revised and enlarged edition), Cambridge, Mass., Harvard University Press, 272–282.

Quine, W. V., 1981a, 'Predicates, Terms, and Classes', in Quine, W. V., *Theories and Things*, Cambridge, Mass. and London, England, The Belknap Press of Harvard University Press, 164–172.

Quine, W. V., 1981b, 'Predicate Functors Revisited', *Journal of Symbolic Logic* **46**, 649–652.

RUUL, Reports from Uppsala University, Dept. of Linguistics,
Box 527, SE-751 20 UPPSALA, Sweden.
`http://www.ling.uu.se/ruul/`
This volume, nr 34 (September 1999), contains
Three Papers on Computational Syntax and Semantics
by Mats Dahllöf.
ISBN: 91-973737-0-2 (this volume). ISSN: 0280-1337 (*RUUL*).

Printed by *Repro Ekonomikum*, Uppsala.