Technical report CS-2001-05

Modular HPSG

Vlado Kešelj

Department of Computer Science, University of Waterloo,
Waterloo, ON N2L 3G1, Canada,
vkeselj@cs.uwaterloo.ca, http://www.cs.uwaterloo.ca/~vkeselj

Abstract. We present a detailed description of the modular Head-driven Phrase Structure Grammar (HPSG). Although the notion of modularity is known in the area of programming languages, and it is described for context-free grammars (CFG), this is the first attempt of defining modularity for HPSGs.
We describe and formally define modularity for an HPSG-type grammar, and we illustrate its application on an example.

Key words: HPSG, modularity.

1 Introduction

HPSG is currently one of the most important formalisms used in computational linguistics. This unification-based formalism is successfully used in theoretical linguistics to explain many natural-language (NL) phenomena, and it is also used in practical applications.

Although it does scale well up to a certain size of the knowledge base, it appears difficult to achieve a "real-world" coverage with the available methods. HPSG lexicalism supports encapsulation of some fine-grained knowledge at the word level. We propose HPSG modules as a way of encapsulating coarse-grained knowledge at the level of domains (semantic domains, application domains, or similar).

There are two major advantages of using HPSG modules:

First, HPSG modularity makes NL engineering easier in the way object-oriented programming (OOP) makes computer programming easier. Building large and complex NL systems is easier if the problem is divided into well-defined parts, which can be separately developed, tested, and maintained.

And second, HPSG modularity provides a framework for independent development of NL modules in different domains, and their unifying use over the Internet. This is a promising application in the context of current research activity in the area of e-commerce and XML.

2 Related Work

A very readable introduction into the area of unification-based grammars is given in Shieber [12]. The HPSG formalism is described in Carpenter [1], Pollard and Sag [10], and Sag and Wasow [11]. A formal definition suitable for our current experiments in question-answering is given in Kešelj [2].

The notion of composition from the area of programming languages is applied to linguistic formalisms in Wintner [14], where a compositional semantics for unification-based grammars is defined. Wintner [15] presents a compositional semantics for CFGs. It also defines modules for CFGs. Our definition of modules for HPSGs is defined in a similar fashion, following the same motivation.

With respect to the logic of typed feature structures, Penn [9] discusses an algebraic method for extracting sub-signatures from larger signatures. This work is related to the operation of module extraction from a large grammar. We do not discuss module extraction here, but it is relevant to HPSG modularity and it is a part of our future research.

Neumann [5, 7, 6, 8, 4] uses machine learning approaches to automatically extract a corpus-oriented subgrammar. If a corpus is from a specific domain, then the extracted subgrammar will be tuned towards this domain. The HPSG formalism is used, as well as the stochastic lexicalized tree grammars. Similarly to [9], this work is related to module extraction.

Zajac and Jan [16] present a practical approach to modularity of unification-based grammars. The approach is implemented in a system, which is used in several machine-translation applications. The modules are separate executable units, connected within the system in the style of the Unix pipe command. This technique is simple and elegant, but it is also a limited solution. The limitations are mandatory serialization of modules and module isolation. It is not clear that it is always possible to divide a complex natural language processing (NLP) task into simpler sub-tasks that are serializable.

3 HPSG

We define an HPSG to be a standard cyclic unification-based grammar with addition of types, which form a multi-inheritance hierarchy. The types lack signatures, i.e., well-typedness and appropriateness are not defined. The model definition can be found in [2], and we do not repeat it here. In this section, we define HPSG principles, which play an important role in the HPSG modularity.

The definition of an HPSG principle is equivalent to the definition of an HPSG rule ([2]):

Definition 1 (HPSG principle). *An HPSG principle is either phrasal (i.e., non-lexical) HPSG principle or lexical HPSG principle. A phrasal HPSG principle is a tuple* $(X, Y_1, \ldots, Y_n) \in \mathcal{F}^{n+1}$ *(*$n \geq 1$*), denoted as:*

$$X \to Y_1 \ldots Y_n,$$

where X, $Y_1,\ldots,Y_n$ are AVMs. A lexical HPSG principle is a pair $(A, a) \in \mathcal{F} \times \mathsf{Atom}$, denoted as:

$$A \rightarrow a,$$

where A is an AVM, and a is an atom.

The difference between an HPSG rule and an HPSG principle is that an HPSG principle is not used directly in a derivation. The principles are meant to represent relations that are part of HPSG rules. To illustrate this, consider the rule:

$$\begin{bmatrix} np \\ \mathsf{HDTR:}\ \boxed{1} \end{bmatrix} \rightarrow [the]\ \boxed{1}\,[n] \tag{1}$$

This rule is directly applied during generation or parsing. The head principle, which states that the head component of the head daughter is unified with the head component of the mother, can be specified as follows:

$$\begin{bmatrix} \mathsf{HEAD:}\ \boxed{1} \\ \mathsf{HDTR:}\ \begin{bmatrix} \mathsf{HEAD:}\ \boxed{1} \end{bmatrix} \end{bmatrix} \rightarrow [\,]\,[\,]$$

This principle can be applied to rule (1) by unifying it with that rule, and by obtaining the new rule:

$$\begin{bmatrix} np \\ \mathsf{HDTR:}\ \boxed{1} \\ \mathsf{HEAD:}\ \boxed{2} \end{bmatrix} \rightarrow [the]\ \boxed{1} \begin{bmatrix} n \\ \mathsf{HEAD:}\ \boxed{2} \end{bmatrix}$$

If we do not want a principle to be applied to a particular rule, we simply prevent unification between them. For example, the following rule could not be unified with the head principle above:

$$\begin{bmatrix} sen \\ \mathsf{HEAD:}\ \text{-} \end{bmatrix} \rightarrow [s1]\ [s1]$$

Obviously, this operation of applying a principle to a rule can be done manually, ahead of parsing-time, and we do not have to mention principles in the model. However, the principles become important for HPSG modules, since principles from one module are applied to rules of another module. Application of a set of principles to a set of rules is defined in the following way:

Definition 2 (Principles application). *If P is a finite set of principles, and R is a finite set of rules in the same grammar, we define $P \otimes R$ to be the following set of rules:*

$$\begin{aligned} P \otimes R = {} & \{p \sqcup r \,:\, p \in P, r \in R, \ \textit{and}\ p \sqcup r\ \textit{exists}\} \\ & \cup \{r \,:\, r \in R, (\forall p \in P)\ p \sqcup r\ \textit{does not exist}\}. \end{aligned}$$

4 Modularity

The notion of modularity that we present here is partly inspired by the OOP paradigm. In particular, an access control similar to the public/protected/private access in OOP is applied.

An HPSG module incorporates an HPSG. Additionally, we want to be able to merge two modules into a new module. This operation provides an easy way of handling the complexity of developing and maintaining large NL grammars.

An HPSG is a tuple $(\mathsf{Atom}, \mathsf{Feat}, \mathsf{Var}, \mathsf{Type}, \mathsf{Init}, \mathsf{Rule})$, where Atom is an enumerable set of atoms, Feat is a finite set of features (attributes), Var is an enumerable set of variables, Type is a finite, multi-inheritance type hierarchy, Init is a finite set of initial AVMs, and Rule is a finite set of rules. A finite set of principles Prin is also a part of an HPSG module. To define a merge operation of two modules requires that we define how these grammar components combine to form the same components of the resulting grammar. Hence, given two modules

$$\begin{aligned} M_1 &= (\mathsf{Atom}_1, \mathsf{Feat}_1, \mathsf{Var}_1, \mathsf{Type}_1, \mathsf{Init}_1, \mathsf{Rule}_1, \mathsf{Prin}_1), \quad \text{and} \\ M_2 &= (\mathsf{Atom}_2, \mathsf{Feat}_2, \mathsf{Var}_2, \mathsf{Type}_2, \mathsf{Init}_2, \mathsf{Rule}_2, \mathsf{Prin}_2), \end{aligned}$$

we describe how to obtain the result of their merge:

$$M = M_1 \cup M_2 = (\mathsf{Atom}, \mathsf{Feat}, \mathsf{Var}, \mathsf{Type}, \mathsf{Init}, \mathsf{Rule}, \mathsf{Prin}).$$

Atoms. Typical atoms are words of a natural language. The set of atoms is usually the set of all words over a finite alphabet. We assume that both modules are defined over the same set of atoms, i.e., that $\mathsf{Atom}_1 = \mathsf{Atom}_2$, and that $\mathsf{Atom} = \mathsf{Atom}_1 = \mathsf{Atom}_2$. In order words, all atoms are *public.*

Features. The features are divided into two sets: public and private features, i.e., $\mathsf{Feat}_1 = \mathsf{Feat}_1^{pub} \cup \mathsf{Feat}_1^{priv}$ and $\mathsf{Feat}_2 = \mathsf{Feat}_2^{pub} \cup \mathsf{Feat}_2^{priv}$. Two public features from the modules that have the same name represent the same feature in the resulting module. If at least one of them is private, then they are not the same features.

For example, if the feature A is a public feature of the module M_1, and it is a public feature of the module M_2, then

$$\begin{bmatrix} \mathsf{A}\text{: } 1 \end{bmatrix}^{(M_1)} \sqcup \begin{bmatrix} \mathsf{A}\text{: } 1 \end{bmatrix}^{(M_2)} = \begin{bmatrix} \mathsf{A}\text{: } 1 \end{bmatrix}$$

where the superscripts (M_1) and (M_2) denote that the first AVM is from the module M_1 and the second from module M_2.

However, if the feature A is a private feature of M_1 and a public feature of M_2, then we have:

$$\begin{bmatrix} \mathsf{A}\text{: } 1 \end{bmatrix}^{(M_1)} \sqcup \begin{bmatrix} \mathsf{A}\text{: } 1 \end{bmatrix}^{(M_2)} = \begin{bmatrix} \mathsf{A}^{(M_1)}\text{:} & 1 \\ \mathsf{A}\text{:} & 1 \end{bmatrix}$$

Similarly, if the feature A is a private feature of M_1 and a private feature of M_2, then:

$$\begin{bmatrix}\mathsf{A}\text{: } 1\end{bmatrix}^{(M_1)} \sqcup \begin{bmatrix}\mathsf{A}\text{: } 1\end{bmatrix}^{(M_2)} = \begin{bmatrix}\mathsf{A}^{(M_1)}\text{: } 1 \\ \mathsf{A}^{(M_2)}\text{: } 1\end{bmatrix}$$

The resulting sets of features are created in the following way:

$$\begin{aligned}\mathsf{Feat}^{pub} &= \mathsf{Feat}_1^{pub} \cup \mathsf{Feat}_2^{pub}\\ \mathsf{Feat}^{priv} &= \{(f, M_1) \,:\, f \in \mathsf{Feat}_1^{priv}\} \cup \{(f, M_2) \,:\, f \in \mathsf{Feat}_2^{priv}\},\ \text{and}\\ \mathsf{Feat} &= \mathsf{Feat}^{pub} \cup \mathsf{Feat}^{priv}.\end{aligned}$$

We assume that no public features have the form (f, M'), where M' is an HPSG module.

Variables. We assume that $\mathsf{Var}_1 = \mathsf{Var}_2 = \mathsf{Var}$.

Type hierarchy. Similarly to features, the types are divided into public and private types. Two public types with the same name are merged into one type, while two private types from different modules are always different after a merge.

Unlike features, the types do not form just a set, but a type hierarchy. A type hierarchy is a partial order $\mathsf{Type} = (\mathsf{T}, \sqsubseteq)$, with the minimal element $\bot$, representing the most general type, and such that for any two types $t_1, t_2 \in \mathsf{T}$ there is at most one least upper bound for both of them, i.e.,

$$(\forall t_3, t_4 \in \mathsf{T})\ (t_1, t_2 \sqsubseteq t_3)\ \wedge (t_1, t_2 \sqsubseteq t_4)\ \Rightarrow\ (t_3 \sqsubseteq t_4)\ \vee (t_4 \sqsubseteq t_3). \qquad (2)$$

If two types t_1 and t_2 do not have the least upper bound, then we say that they cannot be unified; otherwise, we say that they can be unified and the result of their unification is the least upper bound, denoted as $t_1 \sqcup t_2$.

For each module, the types are divided into private and public types, i.e., $\mathsf{T}_1 = \mathsf{T}_1^{priv} \cup \mathsf{T}_1^{pub}$, and $\mathsf{T}_2 = \mathsf{T}_2^{priv} \cup \mathsf{T}_2^{pub}$. The sets T_1 and T_2 are the type sets of the modules M_1 and M_2, i.e., $\mathsf{Type}_1 = (\mathsf{T}_1, \sqsubseteq_1)$ and $\mathsf{Type}_2 = (\mathsf{T}_2, \sqsubseteq_2)$. The resulting type set is obtained in the same way as the feature set:

$$\begin{aligned}\mathsf{T}^{pub} &= \mathsf{T}_1^{pub} \cup \mathsf{T}_2^{pub}\\ \mathsf{T}^{priv} &= \{(t, M_1) \,:\, t \in \mathsf{T}_1^{priv}\} \cup \{(t, M_2) \,:\, t \in \mathsf{T}_2^{priv}\},\ \text{and}\\ \mathsf{T} &= \mathsf{T}^{pub} \cup \mathsf{T}^{priv}.\end{aligned}$$

We assume that no public types have the form (t, M'), where M' is an HPSG module. The most general type is always public, so the resulting hierarchy will also have the most general type. The resulting partial order, i.e., the resulting type subsumption, is the result of the transitive closure of the union of the relations $\sqsubseteq_1$ and $\sqsubseteq_2$, i.e.:

$$\sqsubseteq = (\sqsubseteq_1 \cup \sqsubseteq_2)^+$$

The resulting relation '$\sqsubseteq$' is not necessarily a partial order, even less it necessarily satisfies condition (2). For this reason, two HPSG modules can be merged only if the resulting type hierarchy $\mathsf{Type} = (\mathsf{T}, \sqsubseteq)$ is well-defined, i.e., it is a partial order, and condition (2) is satisfied.

Initial AVMs. The set of initial AVMs Init is the union of Init_1 and Init_2: $\mathsf{Init} = \mathsf{Init}_1 \cup \mathsf{Init}_2$.

Rules. The rules are divided into private and public rules: $\mathsf{Rule}_1 = \mathsf{Rule}_1^{pub} \cup \mathsf{Rule}_1^{priv}$ and $\mathsf{Rule}_2 = \mathsf{Rule}_2^{pub} \cup \mathsf{Rule}_2^{priv}$. The resulting set of rules is obtained in the following way:

$$\begin{aligned} \mathsf{Rule}^{pub} &= \mathsf{Rule}_1^{pub} \cup \mathsf{Rule}_2^{pub} \\ \mathsf{Rule}^{priv} &= \mathsf{Rule}_1^{priv} \cup \mathsf{Rule}_2^{priv} \\ \mathsf{Rule} &= \mathsf{Rule}^{pub} \cup \mathsf{Rule}^{priv} \end{aligned}$$

Principles. All principles are public, i.e.:

$$\mathsf{Prin} = \mathsf{Prin}_1 \cup \mathsf{Prin}_2$$

The previous discussion can be summarized into the following two definitions, in which we finally precisely define the notion of HPSG module and module merge operation.

Definition 3 (HPSG module). *An HPSG module is a tuple*

$$(\mathsf{Atom}, \mathsf{Feat}^{pub}, \mathsf{Feat}^{priv}, \mathsf{Var}, \mathsf{T}^{pub}, \mathsf{T}^{priv}, \sqsubseteq, \mathsf{Init}, \mathsf{Rule}^{pub}, \mathsf{Rule}^{priv}, \mathsf{Prin}), \qquad (3)$$

such that $\mathsf{Feat}^{pub} \cap \mathsf{Feat}^{priv} = \emptyset$, $\mathsf{T}^{pub} \cap \mathsf{T}^{priv} = \emptyset$, *no elements from* Feat^{pub} *have form* (f, M') *where* M' *is an HPSG module, no elements from* T^{pub} *have form* (t, M')*, and the tuple*

$$(\mathsf{Atom}, \mathsf{Feat}^{pub} \cup \mathsf{Feat}^{priv}, \mathsf{Var}, (\mathsf{T}^{pub} \cup \mathsf{T}^{priv}, \sqsubseteq), \mathsf{Init}, (\mathsf{Prin} \otimes \mathsf{Rule}) \cup \mathsf{Rule}^{priv}) \quad (4)$$

is an HPSG grammar.

We say that grammar (4) is defined by module (3).

Definition 4 (Module merge). *The* merge *of two HPSG modules* M_1 *and* M_2*, where:*

$M_1 =$
$(\mathsf{Atom}_1, \mathsf{Feat}_1^{pub}, \mathsf{Feat}_1^{priv}, \mathsf{Var}_1, \mathsf{T}_1^{pub}, \mathsf{T}_1^{priv}, \sqsubseteq_1, \mathsf{Init}_1, \mathsf{Rule}_1^{pub}, \mathsf{Rule}_1^{priv}, \mathsf{Prin}_1)$
$M_2 =$
$(\mathsf{Atom}_2, \mathsf{Feat}_2^{pub}, \mathsf{Feat}_2^{priv}, \mathsf{Var}_2, \mathsf{T}_2^{pub}, \mathsf{T}_2^{priv}, \sqsubseteq_2, \mathsf{Init}_2, \mathsf{Rule}_2^{pub}, \mathsf{Rule}_2^{priv}, \mathsf{Prin}_2)$

is the HPSG module $M = M_1 \cup M_2$*, where*

$$M = (\mathsf{Atom}, \mathsf{Feat}^{pub}, \mathsf{Feat}^{priv}, \mathsf{Var}, \mathsf{T}^{pub}, \mathsf{T}^{priv}, \sqsubseteq, \mathsf{Init}, \mathsf{Rule}^{pub}, \mathsf{Rule}^{priv}, \mathsf{Prin})$$

satisfying the following conditions:

1. $\mathsf{Atom} = \mathsf{Atom}_1 = \mathsf{Atom}_2$

2. $\mathsf{Feat}^{pub} = \mathsf{Feat}_1^{pub} \cup \mathsf{Feat}_2^{pub}$ *and*

$$\mathsf{Feat}^{priv} = \{(f, M_1) \,:\, f \in \mathsf{Feat}_1^{priv}\} \cup \{(f, M_2) \,:\, f \in \mathsf{Feat}_2^{priv}\}$$

3. $\mathsf{Var}_1 = \mathsf{Var}_2 = \mathsf{Var}$.

4. $\mathsf{T}^{pub} = \mathsf{T}_1^{pub} \cup \mathsf{T}_2^{pub}$ *and*

$$\mathsf{T}^{priv} = \{(t, M_1) \,:\, t \in \mathsf{T}_1^{priv}\} \cup \{(t, M_2) \,:\, t \in \mathsf{T}_2^{priv}\}$$

5. $(\mathsf{T}^{pub} \cup \mathsf{T}^{priv}, \sqsubseteq)$ *is a type hierarchy,*

6. $\mathsf{Init} = \mathsf{Init}_1 \cup \mathsf{Init}_2$

7. $\mathsf{Rule}^{pub} = \mathsf{Rule}_1^{pub} \cup \mathsf{Rule}_2^{pub}$ *and* $\mathsf{Rule}^{priv} = \mathsf{Rule}_1^{priv} \cup \mathsf{Rule}_2^{priv}$*, and*

8. $\mathsf{Prin} = \mathsf{Prin}_1 \cup \mathsf{Prin}_2$

5 Example

We use a toy example to show how HPSG modules are used in an application to the problem of question answering. The details about using HPSG in the problem of question answering can be found in [3]. Here, we give a short description of the method. The problem of question answering is defined in the following way:

> Given a collection of natural-language documents, find an answer to given NL query that is a short substring of one of the documents, and it is found in a relevant context.

This substring to be returned is called an *answer string.* To find an answer string, we first parse the question and obtain its semantic representation in form of an AVM. After finding relevant passages, each passage is parsed using a chart parsing algorithm. Parsing fills the chart with edges: one edge per one successfully recognized word, phrase, or sentence. Each edge contains an AVM, and these AVMs are matched to the question AVM. The best matching AVM determines the phrase that is the answer string.

The NL component is defined by the five HPSG modules:

1. chart module,
2. general syntactic module,
3. general semantic module,
4. question syntactic module, and
5. question semantic module.

We use an example question from TREC-8 [13]:

When was London's Docklands Light Railway constructed? (5)

A classical IR system based on a keyword approach could not find an answer to this question. However it did retrieve a relevant passage. A correct answer string was:

... the opening of the railway in 1987 ... (6)

Chart module is related to chart parsing, and it is not concerned with linguistic knowledge. The span of each edge in the chart is determined by two features FROM and TO, which have integer values. These two features are private, which is an important fact for the operation of matching the query AVM to the passage AVMs. Since there are two different charts for queries and passages, the values of the features FROM and TO cannot be compared, i.e., unified. It is achieved by having both features private, and by creating two different instances of the chart module.

The chart module includes the following principles:

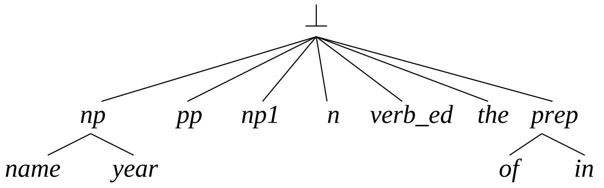
$$\begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{2}\end{bmatrix} \rightarrow \begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{2}\end{bmatrix}$$

$$\begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{2}\end{bmatrix} \rightarrow \begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{3}\end{bmatrix}\begin{bmatrix}\text{FROM:} & \boxed{3}\\ \text{TO:} & \boxed{2}\end{bmatrix}$$

$$\begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{2}\end{bmatrix} \rightarrow \begin{bmatrix}\text{FROM:} & \boxed{1}\\ \text{TO:} & \boxed{3}\end{bmatrix}\begin{bmatrix}\text{FROM:} & \boxed{3}\\ \text{TO:} & \boxed{4}\end{bmatrix}\begin{bmatrix}\text{FROM:} & \boxed{4}\\ \text{TO:} & \boxed{2}\end{bmatrix}$$

In our grammar, there are no rules with more than three daughters, so the principles above cover all rules. The principles introduce the natural relations of spans between daughters and a mother.

General syntactic module describes syntactic properties of words and phrases. It consists of the following type hierarchy:



the following lexical rules:

$$\begin{bmatrix}\textit{name}\\ \text{NUM: sg}\end{bmatrix} \rightarrow \text{London's Docklands Light Railway}$$

$$\begin{array}{rlrl}
[\textit{verb\_ed}] & \rightarrow \text{constructed} & [\textit{the}] & \rightarrow \text{the}\\
[\textit{n}] & \rightarrow \text{opening} & [\textit{of}] & \rightarrow \text{of}\\
[\textit{n}] & \rightarrow \text{railway} & [\textit{in}] & \rightarrow \text{in}\\
[\textit{year}] & \rightarrow \text{1987} & &
\end{array}$$

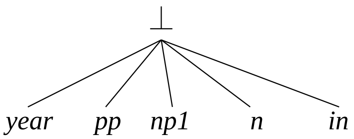
and the following phrasal rules:

$$\begin{array}{rlrl}
[\textit{pp}] & \rightarrow [\textit{prep}][\textit{np}] & [\textit{np1}] & \rightarrow [\textit{n}]\\
[\textit{np1}] & \rightarrow [\textit{np1}][\textit{pp}] & [\textit{np}] & \rightarrow [\textit{the}][\textit{np1}]
\end{array}$$

All features and all types are public, and there are no principles. This module recognizes words and some phrases of given query (5) and string (6). Due to prepositional phrase attachment ambiguity, it produces two parses for the passage. The phrase 'in 1987' can be attached to the word 'railway' as well as to the word 'opening'.

General semantic module describes semantics of words and phrases. We use this general module to disambiguate prepositional phrase attachment ambiguity using the feature 'EVENT'. Namely, if we assume that 'opening' is an event, that 'railway' is not an event, and that a prepositional phrase of the form 'in *year*' can only modify events, then the previous ambiguity is resolved.

All types are public and the type hierarchy is:



The module does not contain any rules, and the set of principles is the following:

$$\begin{bmatrix} \mathit{pp} \\ \mathsf{SEM{:}OBJ{:}}\ \boxed{1} \end{bmatrix} \rightarrow [\mathit{of}]\ \boxed{1}\,[\mathit{np}]$$

$$\begin{bmatrix} \mathit{pp} \\ \mathsf{SEM{:}DATE{:}}\ \boxed{1} \end{bmatrix} \rightarrow [\mathit{prep}] \begin{bmatrix} \mathit{np} \\ \mathsf{SEM{:}DATE{:}}\ \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} \mathit{np1} \\ \mathsf{SEM{:}EVENT{:}}\ \boxed{1} \end{bmatrix} \rightarrow \begin{bmatrix} \mathit{n} \\ \mathsf{SEM{:}EVENT{:}}\ \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} \mathit{np1} \\ \mathsf{SEM{:}EVENT{:}}\ - \end{bmatrix} \rightarrow \begin{bmatrix} \mathit{n} \\ \mathsf{SEM{:}EVENT{:}}\ - \end{bmatrix} \begin{bmatrix} \mathit{pp} \\ \mathsf{SEM{:}DATE{:}}\ - \end{bmatrix}$$

$$\begin{bmatrix} \mathit{np1} & \\ \mathsf{SEM{:}} & \begin{bmatrix} \mathsf{EVENT{:}} & \boxed{1} \\ \mathsf{DATE{:}} & \boxed{2} \end{bmatrix} \end{bmatrix} \rightarrow \begin{bmatrix} \mathit{n} \\ \mathsf{SEM{:}EVENT{:}}\ \boxed{1}[\,] \end{bmatrix} \begin{bmatrix} \mathit{pp} \\ \mathsf{SEM{:}DATE{:}}\ \boxed{2} \end{bmatrix}$$

$$\begin{bmatrix} \mathsf{SEM{:}EVENT{:}DESC{:}}\ \text{open} \end{bmatrix} \rightarrow \text{opening}$$

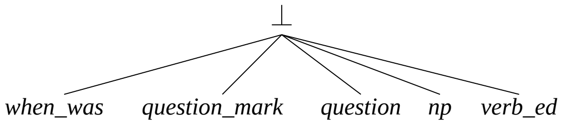
$$\begin{bmatrix} \mathsf{SEM{:}EVENT{:}}\ - \end{bmatrix} \rightarrow \text{railway}$$

$$\begin{bmatrix} \mathsf{SEM{:}DATE{:}}\ 1987 \end{bmatrix} \rightarrow 1987$$

$$\begin{bmatrix} \mathsf{SEM{:}EVENT{:}DESC{:}}\ \text{construct} \end{bmatrix} \rightarrow \text{constructed}$$

Now, the prepositional phrase attachment is resolved since the word 'railway' is not an event, and thus it will not allow a modifier with a feature value of 'DATE' set, i.e., different than '–'.

Question syntactic module defines higher-level question topology. It has one feature 'NUM', all its types are public, and the type hierarchy is:



All rules are public and the following is the set of rules:

$$\begin{bmatrix} \textit{when\_was} \end{bmatrix} \rightarrow \text{When was}$$

$$\begin{bmatrix} \textit{question\_mark} \end{bmatrix} \rightarrow \text{?}$$

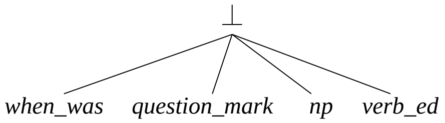
$$\begin{bmatrix} \textit{question} \end{bmatrix} \rightarrow \begin{bmatrix} \textit{when\_was} \end{bmatrix} \begin{bmatrix} \textit{np} \\ \textsf{NUM}: \text{sg} \end{bmatrix} \begin{bmatrix} \textit{verb\_ed} \end{bmatrix} \begin{bmatrix} \textit{question\_mark} \end{bmatrix}$$

Using this module, we can completely parse the question.

Question semantic module defines specific question semantics. We define question semantics in an SQL-like style using '**SELECT**' and '**WHERE**' attributes. We use the following set of public attributes:

$$\{\textsf{SEM}, \textsf{SELECT}, \textsf{WHERE}, \textsf{DATE}, \textsf{OBJ}\}$$

All types are public and the following is the type hierarchy:



The module does not contain any rules, and it contains one principle:

$$\begin{bmatrix} \textsf{SEM}: \begin{bmatrix} \textsf{SELECT}: \boxed{1} \\ \textsf{WHERE}: \boxed{2} \begin{bmatrix} \textsf{DATE}: \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix} \rightarrow$$

$$[\textit{when\_was}]\ \boxed{3}\,[\textit{np}] \begin{bmatrix} \textit{verb\_ed} \\ \textsf{SEM}: \quad \boxed{2} \begin{bmatrix} \textsf{OBJ}: \boxed{3} \end{bmatrix} \end{bmatrix} [\textit{question\_mark}]$$

Module application. We merge all four modules and apply the resulting grammar to query (5). One parse tree is obtained, and the semantic part of the result is the following:

$$\begin{bmatrix} \textit{question} & \\ \textsf{SEM}: & \begin{bmatrix} \textsf{SELECT}: & \boxed{1} \\ \textsf{WHERE}: & \begin{bmatrix} \textsf{EVENT}: & \begin{bmatrix} \textsf{DESC}: \text{construct} \end{bmatrix} \\ \textsf{OBJ}: & \begin{bmatrix} \textit{np} \\ \textsf{NUM}: \text{sg} \end{bmatrix} \\ \textsf{DATE}: & \boxed{1} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

The semantic part of the representation of string (6) is:

$$\begin{bmatrix} np & \\ \mathsf{SEM:} & \begin{bmatrix} \mathsf{EVENT:} & \begin{bmatrix} \mathsf{DESC:}\ \text{open} \end{bmatrix} \\ \mathsf{DATE:} & 1987 \\ \mathsf{OBJ:} & \begin{bmatrix} np \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Now, since the word 'railway' appears in the span of the AVM associated with the feature 'OBJ', we can match the question AVM to the answer AVM.

6 Conclusion and Future Work

We present a first attempt of defining modularity for HPSGs. The notion of an HPSG module is discussed, and precisely defined. Using a small example from a real-world application, we show how it can be used to handle complexity of NLP.

The future work includes:

- defining operation of module extraction from a large grammar, give a domain specification, and
- defining modules for stochastic HPSGs.

Acknowledgments

I wish to thank Dr. Nick Cercone for valuable discussions and comments regarding this work. I also thank Shuly Wintner for a useful discussion, and Gerald Penn and Günter Neumann for pointing me to some relevant work.

References

1. Bob Carpenter. *The Logic of Typed Feature Structures with Applications to Unification-based Grammars, Logic Programming and Constraint Resolution*, volume 32 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, New York, 1992.
2. Vlado Kešelj. Stefy: Java parser for HPSGs, version 0.1. Technical Report CS-99-26, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada, 2000. `ftp://cs-archive.uwaterloo.ca/cs-archive/CS-99-26/`.
3. Vlado Kešelj. Question answering using unification-based grammar. In *Proceedings of the Fourtheenth Canadian Conference on Artificial Intelligence, AI'2001*, Ottawa, Canada, June 2001. To appear.
4. Günter Neumann. Automatic extraction of stochastic lexicalized tree grammars from treebanks. In A. Abeille, editor, *Treebanks: building and using syntactically annotated corpora*. Kluwer. to appear (the book is based on the ATALA workshop on treebanks, Paris, 1999).
`http://www.dfki.uni-sb.de/~neumann/publications/neumann-ref.html`.

5. Günter Neumann. Application of explanation-based learning for efficient processing of constraint-based grammars. In *Proceedings of the Tenth Conference on Artificial Intelligence for Applications*, San Antonio, Texas, March 1994.
6. Günter Neumann. Applying explanation-based learning to control and speeding-up natural language generation. In *Proceedings of ACL/EACL-97*, Madrid, 1997.
7. Günter Neumann. An on-line learning method to speed-up natural language processing. Technical report, DFKI GmbH, 1997.
8. Günter Neumann. Learning stochastic lexicalized tree grammars from HPSG. Technical report, DFKI GmbH, Saarbrücken, 1999.
9. Gerald B. Penn. *The Algebraic Structure of Attributed Type Signatures.* PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, USA, 2000. `http://www.cs.cmu.edu/~gpenn/cards/thesis.html`.
10. Carl J. Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, 1994.
11. Ivan A. Sag and Thomas Wasow. *Syntactic Theory: A Formal Introduction.* CSLI Publications, Stanford, 1999.
12. Stuart M. Shieber. *An Introduction to Unification-Based Approaches to Grammar.* Number 4 in CSLI Lecture Notes. Center for the Study of Language and Information, Stanford University, Stanford, CA, 1986.
13. Ellen M. Voorhees and Dawn M. Tice. The TREC-8 question answering track evaluation. In *Proceedings of the Eight Text REtrieval Conference (TREC-8)*, 1999. `http://trec.nist.gov/pubs/trec8/t8_proceedings.html`.
14. Shuly Wintner. Compositional semantics for linguistic formalisms. In *Proceedings of ACL'99, the 37th Annual Meeting of the Association for Computational Linguistics*, pages 96–103, June 1999.
`http://www.cis.upenn.edu/~shuly/publications/`.
15. Shuly Wintner. Modularized context-free grammars. In *MOL6 — Sixth Meeting on Mathematics of Language*, pages 61–72, Orlando, Florida, July 1999.
16. Rémi Zajac and Amtrup Jan. Modular unification-based parsers. In *IWPT 2000, Sixth International Workshop on Parsing Technologies*, Trento, Italy, 23–25 February 2000. `http://crl.nmsu.edu/~rzajac/index.html`.