Stefy: Java Parser for HPSGs*

Version 0.1

CS-99-26

Vlado Kešelj

Department of Computer Science
University of Waterloo
Waterloo, Ontario, Canada
e-mail: vkeselj@uwaterloo.ca

July 25, 2000

Abstract

This document describes design and implementation of the system Stefy—a Java parser for Head-Driven Phrase Structure Grammars (HPSG). The parser is used in an Internet information retrieval system.

*This work is supported by NSERC.

Acknowledgments

I wish to thank my supervisor Dr. Nick Cercone for valuable discussions and comments regarding this report. This work is supported by the Natural Sciences and Engineering Research Council of Canada (NSERC).

Contents

1 Introduction and Motivation

This report describes version 0.1 of the parser Stefy—a parser for natural languages (NL), implemented in the programming language Java, and based on the HPSG (Head-Driven Phrase Structure Grammar) model for NLs.

The parser is a part of a larger project to implement a natural language processing (NLP) system for Internet information retrieval (IR). This IR task requires Java applets capable of parsing a NL. More details regarding this application are presented in [9].

HPSG. The HPSG formalism for NLP is a high-level unification approach, suitable for studies in theoretical linguistics and strongly influenced by the Prolog paradigm. As any other grammar formalism, it is used to describe syntax of a NL; but, unlike other grammars, HPSG also provides a way of describing semantics simultaneously. Additionally, HPSG is a very flexible model, in which a large number of NL phenomena can be effectively handled ([18]), and whose grammars are robust and can be relatively easily managed. For this reason, the HPSG model is attractive for lower-level, practical applications, i.e., for the area of natural language engineering. More details about the formalism and a precise definition of it, as used in Stefy, are given in subsection 2.1.

Motivation behind Java parser. As mentioned above, this Java parser will be used in Internet IR applications. Some work on developing HPSG parsers in low-level programming languages has been done ([12]). However, we are not aware of any reports about systems implemented in Java.

The additional reasons for using Java for this task include:

- Java supports dynamic class loading and object serialization, which are important features necessary for our concept of *distributed NLP,*

- Java is a good prototyping language, compared to C++ for example, and facilitates easy experimentation with various approaches, which makes this shift in programming language paradigm less drastic and easier to handle.

1.1 Background

HPSG. The HPSG model was formulated in 1986. The main books on the subject are Pollard and Sag 1994 [14] and Sag and Wasow 1999 [18]. Carpenter 1992 [1] gives a precise formal treatment of the formalism. Another approach to the HPSG formalism is given by King 89 and 94 [10].

Parsers. The first HPSG chart parser was introduced by Proudian and Pollard in 1985 [16]. The parser was implemented in Prolog. In 1990, Popowich and Vogel described their HPSG chart parser [15], also developed in Prolog, which had a better coverage of the developments in the HPSG formalism up to that time. Probably, the best-known and the most widely used parser today is the system ALE (Carpenter and Penn 1999 [3]), which is developed in Prolog. The system LiLFeS (Makino et al. 1999 [12]) is an HPSG parser implemented in a low-level language. In Carpenter 1999 [2], a chart parser for probabilistic context-free grammars, implemented in Java, is described. The LKB system (Copestake et al. 1999 [4]) is a grammar and lexicon development environment developed in Lisp. It is designed for use with constraint-based formalisms, more specifically for the use of typed feature structures, such as the HPSG formalism.

Unification. Unification is a built-in feature in the programming language Prolog, but it is not a part of Java. It is an essential notion in the HPSG model, so it is implemented within the project Stefy in form of a Java package called `avm`. The method of unification (at least in the modern sense) is introduced by Robinson in 1965 [17]. Our unification algorithm is a variation of the Huet's algorithm (Huet 1976 [8]). Knight 1989 [11] gives a good survey on the subject of unification. Malouf 1999 [13] treats some efficiency issues regarding graph unification for parsing constraint-based grammars, which are also relevant for the HPSG parsing.

2 Formalism

2.1 HPSG Formalism

The HPSG formalism is a relatively complex notion (compared to context-free grammars, for example), and it can be approached in various ways. One

of the choices that we have to make is the tradeoff between implementational efficiency and efficacy of directly capturing of various linguistic rules. In this section we present our definition of the HPSGs.

This definition is implementation-oriented, so it does not necessarily reflect an ideal way of defining HPSGs in a theoretical-linguistic sense. However, it is more suitable for our task than the existing formalism. The definition presented here is more compact, and provides for an easy implementation in programming languages that do not include a built-in unification mechanism. The formalism is not suitable for development of grammar and lexical resources. Other systems, like ALE [3] and LKB [4], are more suitable for this task. After a grammar or a lexicon is developed in one of those systems, it is translated into a Java description and used in our system.

Before defining the crucial notion of the HPSG formalism—the *head-driven phrase structure grammar*—we incrementally define several simpler notions.

The HPSG formalism is based on the *attribute-value matrices* (AVM), also called *(typed) feature structures.* The AVMs are recursive structures, and an important question is whether we allow cycles in those structures; e.g., can a proper part of a matrix be the matrix itself. Acyclic matrices are simpler to describe theoretically. On the other hand, allowing cycles is less expensive from the implementational point of view and they provide for a more direct representations of some NL phenomena.[1] For this reason, we choose to include cyclic matrices.

Example 1. Some examples of AVMs are given below:

$$\begin{bmatrix} \textit{employee} & \\ \mathsf{ID}: & \text{JQP} \\ \mathsf{NAME}: & \text{John Q. Public} \\ \mathsf{DEPT}: & \begin{bmatrix} \textit{dept} & \\ \mathsf{ID}: & \text{D1} \\ \mathsf{NAME}: & \text{Dept. of CS} \end{bmatrix} \end{bmatrix}$$

[1] E.g., see [1]. For example, the sentence "This sentence is false." requires a cyclic structure for its semantic representation.

This is an example of an AVM containing data about an employee—a typical example from the database domain. The italic identifiers *employee* and *dept* denote *types,* the strings typeset in the sans-serif font and followed by a colon, e.g. ID, denote *attributes,* and the other strings, e.g 'John Q. Public', are called *atoms.*

A list containing two atoms 'a' and 'b', i.e., $\langle \mathrm{a}, \mathrm{b} \rangle$, can be represented as follows:

$$\begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \mathrm{a} \\ \mathsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \mathrm{b} \\ \mathsf{TAIL}: & [\textit{elist}] \end{bmatrix} \end{bmatrix}$$

The important notion of *structure sharing* can be illustrated by the following AVM:

$$\begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \boxed{1} \\ \mathsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \mathsf{HEAD}: & \boxed{1} \\ \mathsf{TAIL}: & [\textit{list}] \end{bmatrix} \end{bmatrix}$$

This AVM can be interpreted as a pattern that matches all lists having at least two elements, and having the first two elements equal. The symbol $\boxed{1}$ denotes an *index,* or a *variable,* and it is used to denote "identical" entities, in a certain sense that will be explained later.

An example of a cyclic structure is the following:

$$\boxed{1} \begin{bmatrix} \textit{employee} & \\ \mathsf{ID}: & \mathrm{NC} \\ \mathsf{NAME}: & \mathrm{Nick\ Cercone} \\ \mathsf{DEPT}: & \begin{bmatrix} \textit{dept} & \\ \mathsf{ID}: & \mathrm{CS} \\ \mathsf{NAME}: & \mathrm{Dept.\ of\ CS} \\ \mathsf{CHAIR}: & \boxed{1} \end{bmatrix} \end{bmatrix}$$

It describes an employee, who is the chair of the department he belongs to.

Now, we can more formally introduce the notions illustrated above. Let A_t denote a finite set of *attributes,* let A denote an infinite enumerable set of *atoms,* and let V denote an infinite enumerable set of *variables.* We assume that these sets are disjoint.

A set of types is a finite lattice:

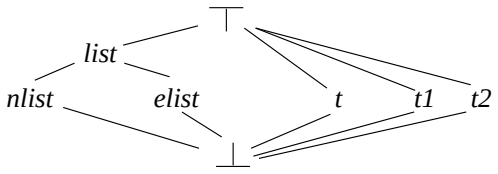
Definition 1 (Lattice of types) *A* lattice of types *is a non-empty finite set T with a partial ordering $\leq$, such that for every pair $t, u \in T$ there is a unique smallest element $t \vee u$ that is greater than t and u, and there is a unique largest element $t \wedge u$ that is smaller than t and u.*

As a consequence, there exist the minimal and the maximal element of the set T, and they are denoted $\bot$ and $\top$.

We assume that the sets A_t, A, V, and T are disjoint.

The type $\bot$ is specific in our application in sense that it is not considered a "valid type;" rather, it is used to denote inconsistencies in unification operations.

Example 2. A lattice of types can be represented by a diagram, e.g.



represents a lattice of types, where $nlist \vee elist = list$, $elist \wedge t = \bot$, and so on. Types *nlist* and *elist* can be used to represent non-empty and empty lists, as already illustrated in the previous example.

The first step towards defining AVMs is the definition of a simpler kind of AVM, which we call *basic AVM.*

Definition 2 (Basic AVM) *A* basic AVM *over the set of attributes A_t, the set of atoms A, the set of variables V, and the lattice of types T (or simply, over A_t, A, V, and T) is any atom, variable, or any tuple of the form*

$$(t, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n)),$$

where $n \geq 0$, $t \in T \setminus \{\bot\}$, $a_i \in A_\mathrm{t}$, $v_i \in V$, *and* $a_i \neq a_j$ *for* $i \neq j$. *The set of all basic AVMs over* A_t, A, V, *and* T *is denoted* $\overline{M}_{A_\mathrm{t},A,V,T}$, *or simply* $\overline{M}$.

We assume that the set of tuples defined in this way does not have any elements in common with the sets A_t, A, V, *or* T.

A basic AVM that is a tuple $(t, (a_1, v_1), (a_2, v_2), \ldots, (a_n, v_n))$ is denoted

$$\begin{bmatrix} t & \\ a_1 & v_1 \\ a_2 & v_2 \\ \vdots & \vdots \\ a_n & v_n \end{bmatrix}$$

If the type is not explicitly denoted, the type $\top$ is assumed. For instance, $[\,]$ represents the basic AVM: $(\top)$.

Definition 3 (System of equations) *Given a set of basic AVMs* $\overline{M}$, *a* system of equations *(or a* system *for short) is any finite relation* $\sigma \subset V \times \overline{M}$.

Definition 4 (Satisfiable system) *A system* σ *is* satisfiable *if there exists an equivalence relation* $\equiv$ *on* $\overline{M}$, *such that* $\sigma \subseteq \equiv$, *and for all* $\overline{m} \equiv \overline{m}'$ *one of the following holds:*

1. $\overline{m} \in V$ *or* $\overline{m}' \in V$,

2. $\overline{m}, \overline{m}' \in A$ *and* $\overline{m} = \overline{m}'$, *or*

3. $\overline{m} = (t, (a_1, v_1), \ldots, (a_n, v_n))$, $\overline{m}' = (t', (a'_1, v'_1), \ldots, (a'_{n'}, v'_{n'}))$, $a_i, a'_i \in A_\mathrm{t}$, $v_i, v'_i \in V$, $a_i = a'_j \Rightarrow v_i \equiv v'_j$; *and there is a type* $t_0 \in T \setminus \{\bot\}$ *such that for all basic AVMs* $\overline{m}'' = (t'', (a''_1, v''_1), \ldots, (a''_{n''}, v''_{n''}))$:

$$\overline{m} \equiv \overline{m}'' \Rightarrow t'' \geq t_0.$$

If σ is a satisfiable system, then there exists the smallest equivalence relation $\equiv$ with the above properties, and we denote it $\equiv_\sigma$.

Example 3. A simple example of an unsatisfiable system is:

$$\{(v_1, \mathrm{a}), (v_2, \mathrm{b}), (v_1, v_2)\}$$

where a and b are two distinct atoms.

Definition 5 (AVM) *An* AVM *is any pair* (v, σ) *of a distinguished variable* v *and a satisfiable system* σ*. The set of all AVMs is denoted* $M_{A_{\mathrm{t}},A,V,T}$*, or simply* M*.*

Definition 6 (Domain of variables) *For any basic AVM* $\overline{m}$*, we define the* domain of variables *of* $\overline{m}$*,* $D_{\mathrm{v}}(\overline{m})$*, to be:*

$$D_{\mathrm{v}}(\overline{m}) = \begin{cases} \emptyset, & \text{if } \overline{m} \in A; \\ \{\overline{m}\}, & \text{if } \overline{m} \in V; \\ \{v_1, \ldots, v_n\}, & \text{if } \overline{m} = (t, (a_1, v_1), \ldots, (a_n, v_n)). \end{cases}$$

For a system σ*, the* domain of variables *of* σ *is*

$$D_{\mathrm{v}}(\sigma) = \bigcup_{(v,\overline{m})\in\sigma} (\{v\} \cup D_{\mathrm{v}}(\overline{m})).$$

For any AVM $m = (v, \sigma)$*, we define the* domain of variables *of* m*,* $D_{\mathrm{v}}(m)$*, to be:*

$$D_{\mathrm{v}}(m) = \{v\} \cup D_{\mathrm{v}}(\sigma).$$

Intuition behind the domain of variables is that a domain of variables includes all variables that are part of an AVM. Two AVMs are "independent" if their domains of variables are disjoint.

Since any system σ is finite, $D_{\mathrm{v}}(m)$ is a finite set for any system or any AVM. It is also finite for any basic AVM.

Example 4. A frequently used structure in HPSGs is a list. For instance, $\langle a_1, a_2 \rangle$ is a list of two atoms $a_1, a_2 \in A$. Using the type hierarchy given in the previous example, the list $\langle a_1, a_2 \rangle$ can be represented in the standard HPSG notation as:

$$\left[\begin{array}{ll} \mathit{nlist} & \\ \textsf{HEAD}: & a_1 \\ \textsf{TAIL}: & \left[\begin{array}{ll} \mathit{nlist} & \\ \textsf{HEAD}: & a_2 \\ \textsf{TAIL}: & \left[\ \mathit{elist}\ \right] \end{array}\right] \end{array}\right].$$

A list of three elements $\langle a_1, a_2, a_3 \rangle$ is represented as:

$$\begin{bmatrix} \textit{nlist} & \\ \textsf{HEAD}: & a_1 \\ \textsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \textsf{HEAD}: & a_2 \\ \textsf{TAIL}: & \begin{bmatrix} \textit{nlist} & \\ \textsf{HEAD}: & a_3 \\ \textsf{TAIL}: & \textit{elist} \end{bmatrix} \end{bmatrix} \end{bmatrix},$$

and so on.

According to Definition 5, the AVM representing the list $\langle a_1, a_2 \rangle$ (on the previous page) is a pair (v_1, σ), where

$$\begin{aligned} \sigma \;=\; & \{(v_1, (\textit{nlist}, (\textsf{HEAD}, v_2), (\textsf{TAIL}, v_3))), (v_2, a_1), \\ & (v_3, (\textit{nlist}, (\textsf{HEAD}, v_4), (\textsf{TAIL}, v_5)), (v_4, a_2), (v_5, (\textit{elist}))\}. \end{aligned}$$

The domain of variables of this AVM is $D_{\mathrm{v}}((v_1, \sigma)) = \{v_1, v_2, v_3, v_4, v_5\}$.

If we want to emphasize the variable indices in the structure of AVM, we can use the following representation:

$$\boxed{1} \begin{bmatrix} \textit{nlist} & \\ \textsf{HEAD}: & \boxed{2}\, a_1 \\ \textsf{TAIL}: & \boxed{3} \begin{bmatrix} \textit{nlist} & \\ \textsf{HEAD}: & \boxed{4}\, a_2 \\ \textsf{TAIL}: & \boxed{5} \left[\textit{elist} \right] \end{bmatrix} \end{bmatrix}$$

The indices of the form $\boxed{i}$ are used to denote variables v_i. However, we commonly use variable indices only when necessary; i.e., to indicate structure sharing.

Definition 7 (HPSG rule) *If $m \in M$ and $v_1, \ldots, v_n \in V$ $(n \geq 1)$, then*

$$m \rightarrow v_1 v_2 \ldots v_n$$

is an HPSG non-lexical rule.

Let Σ be a distinguished subset of A, which we call alphabet, *and its elements are called* letters *or* terminals. *For any $m \in M$ and $a \in \Sigma$,*

$$m \to a$$

is an HPSG lexical rule.

Lexical and non-lexical HPSG rules are called HPSG rules.

We define the domain of variables *for HPSG rules in the following way:*

$$D_{\mathrm{v}}(m \to v_1 v_2 \ldots v_n) = D_{\mathrm{v}}(m) \cup \{v_1, v_2, \ldots, v_n\}$$

and

$$D_{\mathrm{v}}(m \to a) = D_{\mathrm{v}}(m).$$

If $f : V \to V$ is a bijection, then a *variable renaming* defined by f is a transformation that replaces each variable v with its image $f(v)$. Let r be a rule, let $D_{\mathrm{v}}(r) = D$, and let D_1 be a finite set of variables such that $|D| = |D_1|$. Then, we can always find a variable renaming such that after we rename all variables in r, and obtain a new rule r_1, $D_{\mathrm{v}}(r_1)$ becomes D_1.

Definition 8 (HPSG) *An HPSG is any tuple $(\Sigma, A_{\mathrm{t}}, A, V, T, S, P)$, where Σ is an alphabet, A_{t} is a set of attributes, A is a set of atoms ($\Sigma \subseteq A$), V is a set of variables, T is a lattice of types, S is the initial AVM ($S \in M$), and P is a finite set of HPSG rules.*

Definition 9 (HPSG derivation and its domain of variables) *Let $(\Sigma, A_{\mathrm{t}}, A, V, T, S, P)$ be an HPSG. The* HPSG derivations *have the form $m \Rightarrow \alpha$, where $m \in M$ and $\alpha \in (V \cup \Sigma)^+$, and they are defined recursively in the following way:*

1. *If $S = (v, \sigma)$, then $S \Rightarrow v$ is a derivation. The domain of variables of this derivation is defined to be $D_{\mathrm{v}}(S \Rightarrow v) = D_{\mathrm{v}}(S)$.*

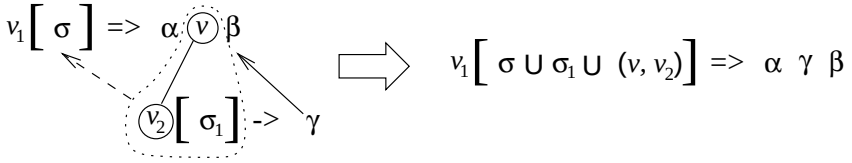
2. *If*

   - *$S_1 \Rightarrow \alpha v \beta$ is a derivation, where $v \in V$, $S_1 = (v_1, \sigma) \in M$, and $\alpha, \beta \in (V \cup \Sigma)^*$,*
   - *$m_1 \to \gamma$ is a rule in P, or is obtained from a rule in P by variable renaming, where $m_1 = (v_2, \sigma_1)$; so that the variable domains of $S_1 \Rightarrow \alpha v \beta$ and $m_1 \to \gamma$ are disjunct, and*

- $S_2 = (v_1, \sigma \cup \sigma_1 \cup (v, v_2))$ *is an AVM,*

then $S_2 \Rightarrow \alpha\gamma\beta$ *is a derivation. Its domain of variables is defined to be* $D_{\mathrm{v}}(S_2 \Rightarrow \alpha\gamma\beta) = D_{\mathrm{v}}(S_1 \Rightarrow \alpha v\beta) \cup D_{\mathrm{v}}(m_1 \rightarrow \gamma)$.

We can illustrate the recursive step in the previous definition with the following diagram:



Example 5. Let

$$\begin{bmatrix} t \\ \textsf{RPREF}: \ \alpha \\ \textsf{SUFF}: \ \ \beta \end{bmatrix}$$

be a form of AVM, which we use to represent a list of atoms. The lists α and β are used to store a reversed prefix and corresponding (non-reversed) suffix of the represented list. For instance, the list $\langle a_1, a_2, a_3 \rangle$ can be represented in the following ways:

$$\begin{bmatrix} t \\ \textsf{RPREF}: \ \langle\rangle \\ \textsf{SUFF}: \ \ \langle a_1, a_2, a_3 \rangle \end{bmatrix}, \qquad \begin{bmatrix} t \\ \textsf{RPREF}: \ \langle a_1 \rangle \\ \textsf{SUFF}: \ \ \langle a_2, a_3 \rangle \end{bmatrix},$$

$$\begin{bmatrix} t \\ \textsf{RPREF}: \ \langle a_2, a_1 \rangle \\ \textsf{SUFF}: \ \ \langle a_3 \rangle \end{bmatrix}, \quad \text{and} \quad \begin{bmatrix} t \\ \textsf{RPREF}: \ \langle a_3, a_2, a_1 \rangle \\ \textsf{SUFF}: \ \ \langle\rangle \end{bmatrix}.$$

We need two rules to derive one list representation from any other equivalent representation: One rule to move an atom from a reversed prefix to the suffix, and another rule to move an atom from a suffix to the reversed prefix. We illustrate just one of the rules, since the other is analogous. First, using the

standard HPSG notation, the rule can be represented as follows:

$$\boxed{1}\begin{bmatrix} t \\ \\ \mathsf{RPREF}\colon\ \boxed{2}\begin{bmatrix} nlist \\ \mathsf{HEAD}\colon\ \boxed{4} \\ \mathsf{TAIL}\colon\ \boxed{5}\ [list] \end{bmatrix} \\ \\ \mathsf{SUFF}\colon\ \boxed{3}\ [list] \end{bmatrix} \rightarrow \boxed{6}\begin{bmatrix} t \\ \mathsf{RPREF}\colon\ \boxed{5} \\ \\ \mathsf{SUFF}\colon\ \boxed{7}\begin{bmatrix} nlist \\ \mathsf{HEAD}\colon\ \boxed{4} \\ \mathsf{TAIL}\colon\ \boxed{3} \end{bmatrix} \end{bmatrix}$$

Now we can see how the rule is expressed according to Definition 7: It is a rule $m \rightarrow v_6$, where $m = (v_1, \sigma)$ is the AVM defined by the following satisfiable system:

$$\begin{aligned} \sigma = \ \{\ & (v_1, (t, (\mathsf{RPREF}, v_2), (\mathsf{SUFF}, v_3))), \\ & (v_2, (nlist, (\mathsf{HEAD}, v_4), (\mathsf{TAIL}, v_5))), \\ & (v_3, (list)), (v_5, (list)), \\ & (v_6, (t, (\mathsf{RPREF}, v_5), (\mathsf{SUFF}, v_7))), \\ & (v_7, (nlist, (\mathsf{HEAD}, v_4), (\mathsf{TAIL}, v_3)))\}. \end{aligned}$$

These two rules are used in the proof of Theorem 1.

Definition 10 (Language generated by HPSG)
Let $(\Sigma, A_t, A, V, T, S, P)$ be an HPSG. The language generated by this grammar is the set:

$$\{w \in \Sigma^* : S \Rightarrow w \textit{ is an HPSG derivation }\}.$$

The HPSG formalism does not give us any new class of languages. It is equivalent to Turing machine, i.e., the following theorem holds:

Theorem 1 (Equivalence to Turing machines) *The class of languages generated by HPSGs is equal to the class of recursively enumerated languages.*

Proof. First, we prove that each language generated by an HPSG is recursively enumerable. By finding all HPSG derivations consisting of no more than n steps given in Definition 9, and incrementally increasing n, we can output all words of the language using a Turing machine. The only nontrivial

routine is verification that a system of equations is satisfiable. We will see in subsection 2.3 how it can be done effectively. This completes one direction of the proof.

Second, we prove that any recursively enumerable language can be generated using an HPSG. It is known that any recursively enumerable language can be generated by an unrestricted grammar in the Chomsky hierarchy (i.e., a type 0 grammar; e.g., see [7]). Hence, it is sufficient to show that any unrestricted grammar can be simulated by an HPSG. Using the AVMs shown in the previous examples, we simulate unrestricted-grammar derivations by representing their sentential forms as lists of terminals and non-terminals. Hence, we include all terminals and non-terminals of the unrestricted grammar in the set of atoms of the HPSG. The sets of atoms, attributes, variables, and the basic-AVM tuples may not be disjoint, but this problem can be easily solved by renaming. If S is the start symbol of the unrestricted grammar, then the initial AVM of the HPSG grammar is:

$$\begin{bmatrix} t \\ \textsf{RPREF}: \langle\rangle \\ \textsf{SUFF}: \quad \langle S \rangle \end{bmatrix}$$

The symbol $\langle\rangle$ denotes the AVM $[\mathit{elist}]$. The rules from the previous example are included, so symbols can be moved orderly from reversed prefix to suffix and vice versa. For each unrestricted rule, for example $ABC \rightarrow XY$, we introduce an HPSG rule:

$$\begin{bmatrix} t \\ \textsf{RPREF}: \boxed{1}\ [\mathit{list}] \\ \textsf{SUFF}: \quad \langle A, B, C, \boxed{2}\ [\mathit{list}] \rangle \end{bmatrix} \rightarrow \begin{bmatrix} t \\ \textsf{RPREF}: \boxed{1} \\ \textsf{SUFF}: \quad \langle X, Y, \boxed{2} \rangle \end{bmatrix}$$

In order to produce a word at the end of a derivation, for each terminal a of the unrestricted grammar, we introduce the following three HPSG rules:

$$\begin{bmatrix} t \\ \textsf{RPREF}: \langle\rangle \\ \textsf{SUFF}: \quad \langle a, \boxed{1}\ [\mathit{list}] \rangle \end{bmatrix} \rightarrow \begin{bmatrix} t1 \\ \textsf{TERM}: a \end{bmatrix} \begin{bmatrix} t \\ \textsf{RPREF}: \langle\rangle \\ \textsf{SUFF}: \quad \boxed{1} \end{bmatrix}$$

$$\begin{bmatrix} t \\ \textsf{RPREF}: \langle\rangle \\ \textsf{SUFF}: \quad \langle a \rangle \end{bmatrix} \rightarrow \begin{bmatrix} t1 \\ \textsf{TERM}: a \end{bmatrix}$$

$$\begin{bmatrix} t1 \\ \mathsf{TERM}\colon\ a \end{bmatrix} \rightarrow a$$

The last rule is the lexical rule. HPSG constructed in this way generates the same language as the starting unrestricted grammar. ■

2.2 Equivalent AVMs

Intuitively, the variables in AVMs are used to connect various parts, and this connecting can be done in various ways for one conceptually unique AVM.

Example 6. The AVM

$$\begin{bmatrix} \mathsf{A}\colon\ \mathrm{a} \\ \mathsf{B}\colon\ \mathrm{b} \end{bmatrix}$$

can be represented in the following two equivalent ways:

$$(v_1, \{(v_1, (\top, (\mathsf{A}, v_2), (\mathsf{B}, v_3))), (v_2, \mathrm{a}), (v_3, \mathrm{b}), (v_4, v_1)\}),$$

and

$$(v_1, \{(v_1, (\top, (\mathsf{A}, v_2))), (v_2, \mathrm{a}), (v_3, \mathrm{b}), (v_4, v_1), (v_4, (\top, (\mathsf{B}, v_3)))\}).$$

(The variable v_4 is redundant in the first case.)

Let us formally define the equivalence relation:

Definition 11 (Equivalent systems and equivalent AVMs)
Two systems σ_1 and σ_2 are equivalent if for any other system σ_3, both systems $\sigma_1 \cup \sigma_3$ and $\sigma_2 \cup \sigma_3$ are either satisfiable or not satisfiable in the same time.

Two AVMs (v_1, σ_1) and (v_2, σ_2) are equivalent, if $v_1 \equiv_{\sigma_1} v_2$ and the systems σ_1 and σ_2 are equivalent.

Two consequences of the above definition are: any two unsatisfiable systems are equivalent, and any unsatisfiable system is not equivalent to any satisfiable system.

We have defined the relation $\equiv_\sigma$ immediately after definition 4. If satisfiable systems σ_1 and σ_2 are equivalent, then the relations $\equiv_{\sigma_1}$ and $\equiv_{\sigma_2}$ are equal. Otherwise, if a pair $(\overline{m}_1, \overline{m}_2)$ belongs to one and not the other

relation, then we can find two variables v_1 and v_2 such that $v_1 \not\equiv_{\sigma_1} v_2$ and $v_1 \not\equiv_{\sigma_2} v_2$, and two distinct atoms a and b, and define:

$$\sigma_3 = \{(v_1, \mathrm{a}), (v_2, \mathrm{b}), (v_1, \overline{m}_1), (v_2, \overline{m}_2)\}$$

According to the above definition, the systems σ_1 and σ_2 are not equivalent, since exactly one of the systems $\sigma_1 \cup \sigma_3$ and $\sigma_2 \cup \sigma_3$ is satisfiable. Hence, the definition of equivalent AVMs is symmetric, i.e., $v_1 \equiv_{\sigma_1} v_2$ if and only if $v_1 \equiv_{\sigma_2} v_2$, for equivalent systems σ_1 and σ_2.

These two relations (system equivalence and AVM equivalence) partition the set of all systems and the set of all AVMs into equivalence classes. Note that all unsatisfiable systems form one class in the set of systems. Each other class of systems can be put in a pair with a distinguished variable, and such a pair is associated with a class of AVMs. In the previous example, we saw two equivalent AVMs. The former AVM does not "divide," the matrix, and we consider that to be a good property. The later AVM does divide the matrix. We distinguish the former AVM, and call it a *compact AVM.* In each equivalence class of AVMs, there is at least one compact AVM.

Definition 12 (Solved system and compact AVM)
A satisfiable system σ is called a solved system *if the following properties are satisfied:*

- *σ has the functional property, i.e., $(x, y) \in \sigma \wedge (x, z) \in \sigma \Rightarrow y = z$,*
- *σ is anti-reflexive, i.e., $(x, x) \notin \sigma$ for all x, and*
- *if σ^+ is the transitive closure of σ, then σ^+ is antisymmetric; i.e., for all variables v_1 and v_2, $(v_1, v_2) \in \sigma^+$ and $(v_2, v_1) \in \sigma^+$ implies $v_1 = v_2$ (i.e., implies a contradiction).*

An AVM (v, σ) is called a compact AVM *if σ is a solved system.*

The compact AVMs are important, since they are easier to represent and more efficient to use in our implementation.

2.3 Unification Algorithm

The standard unification algorithm is reduced to the algorithm for verifying that a system is satisfiable. The problem of unifying two AVMs $m = (v, \sigma)$ and $m_1 = (v_1, \sigma_1)$ is reduced to verifying that the system $\sigma_2 = \sigma \cup \sigma_1 \cup \{(v, v_1)\}$ is satisfiable. The result of the unification is the AVM (v, σ_2). The result could be specified in other ways as well, e.g., (v_1, σ_2), but the resulting AVMs would be equivalent to (v, σ_2). In the implementation, we prefer that the result is in a compact form.

Algorithm IsSatisfiable is given for the problem: Is a system σ satisfiable? In addition to a Boolean output, the algorithm returns a solved system equivalent to σ if the system σ is satisfiable. We use the well-known UNION-FIND structure for representing disjoint sets (e.g., see [5]). POP and PUSH are the standard stack operations used on σ. The algorithm is based on the Huet's unification algorithm [8]. The basic idea is to keep all equivalent variables ($\equiv_\sigma$) in one set, and use an array f to associate each set representative, which is a variable, with a basic AVM tuple or the NULL constant.

Algorithm: **IsSatisfiable**

Input:	σ	input system
Output	**true** or **false**	Is σ satisfiable?
	σ_s	defined if σ is satisfiable; solved and equivalent system to σ

1. **For each** $v \in D_{\mathrm{v}}(\sigma)$ **do** MAKE-SET(v), $f[v] \leftarrow$ NULL
2. **While** $\sigma \neq \emptyset$ **do**
3. $\quad (v, x) \leftarrow \mathrm{POP}(\sigma)$
4. $\quad$ **If** $x \in V$ **then**
5. $\quad\quad f_1 \leftarrow f[\text{FIND-SET}(x)]$
6. $\quad\quad f_2 \leftarrow f[\text{FIND-SET}(v)]$
7. $\quad\quad v \leftarrow \mathrm{UNION}(v, x)$
8. $\quad\quad f[v] \leftarrow f_2$
9. $\quad$ **Else**
10. $\quad\quad f_1 \leftarrow x$
11. $\quad\quad v \leftarrow \text{FIND-SET}(v)$
12. $\quad$ **If** $f[v] = f_1$ **or** $f_1 =$ NULL **then continue**
13. $\quad$ **Else If** $f[v] =$ NULL **then** $f[v] \leftarrow f_1$

14. **Else If** $f_1 \in A$ **or** $f[v] \in A$ **then Return false**
15. **Else**
16. Let $f[v] = (t, (a_1, v_1), \ldots, (a_n, v_n))$ and $f_1 = (t', (a'_1, v'_1), \ldots, (a'_{n'}, v'_{n'}))$
17. $t'' \leftarrow t \wedge t'$
18. **If** $t'' = \bot$ **then Return false**
19. $\{a''_1, \ldots, a''_{n''}\} \leftarrow \{a_1, \ldots, a_n\} \cup \{a'_1, \ldots, a'_{n'}\}$
20. **For each** a''_i in $\{a''_1, \ldots, a''_{n''}\}$ **do**
21. **If** $a''_i = a_j = a'_k$ for some a_j and a'_k **then**
22. PUSH$(\sigma, (v_j, v'_k))$
23. $v''_i \leftarrow v_j$
24. **Else If** $a''_i = a_j$ for some a_j **then** $v''_i \leftarrow v_j$
25. **Else** ($a''_i = a'_k$ for some a'_k) $v''_i \leftarrow v'_k$
26. $f[v] \leftarrow (t'', (a''_1, v''_1), \ldots, (a''_{n''}, v''_{n''}))$
27. $\sigma_s \leftarrow \emptyset$
28. **For each** $v \in D_{\mathrm{v}}(\sigma)$ **do**
29. **If** FIND-SET$(v) = v$ **then** PUSH$(\sigma_s, (v, f[v]))$
30. **Else** PUSH$(\sigma_s, (v,$ FIND-SET$(v)))$
31. **Return true**

In order to determine the running-time complexity, we have to make some assumptions about the size of the input. The input system σ is a set of ordered pairs of a variable and a basic AVM. The size of a basic AVM is a constant if it is an atom or a variable, or it is proportional to the number of pairs in it if it is a tuple; i.e., it is $a + bn$ where a and b are constants, and n is the size of the tuple. We assume that the size of one pair of a variable and a basic AVM in σ is the sum of the sizes of its components, and the size of a system is the sum of the sizes of all such pairs.

The loop 2–26 is repeated $O(n)$ times, where n is the size of the input (not to be confused with n in the algorithm). This is because in each iteration of the loop one pair is popped out of σ (step 3). On the other side, new pairs are pushed in step 22, but each such push reduces the total size of all basic AVMs by one, because two pairs (a_j, v_j) and (a'_k, v'_k) are replaced by one pair (a''_i, v''_i). Hence, there can be at most $O(n)$ pushes.

The loop 28–30 is repeated $O(n)$ times, as well.

There is a finite number of attributes in an HPSG, i.e., a constant number of attributes, so the number of iterations in loop 20–25 is $O(1)$ (within the

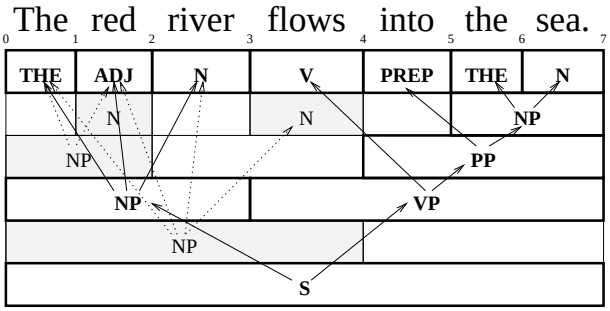


Figure 1: Chart example

outside loop).

The amortized complexity of the operations FIND-SET and UNION is $O(m \cdot \alpha(m))$ for m operations, where $\alpha(m)$ is the very slowly-growing inverse of Ackermann's function [5].

Hence the complexity of our algorithm is $O(n \cdot \alpha(n))$, or $O(na \cdot \alpha(na))$ if a is the number of attributes.

2.4 Parsing Algorithm

The parsing problem in HPSGs is finding all AVMs m such that $m \Rightarrow w$ is an HPSG derivation, given a word $w \in \Sigma^*$. We use a chart parsing algorithm.

All chart parsing algorithms are based on a table called *chart,* or a *well-formed substring table.* A chart contains all complete or incomplete constituents parsed so far. The constituents are physically connected, i.e., they are substrings of the sentence, and the chart provides an easy access to all constituents starting or ending at a certain position. Use of a chart improves the parser performance in terms of time and space: Time spent on parsing components is reused, and space occupied by the components is reused by including daughters in a mother just by their references. An example of a chart is given in figure 1.

In a typical chart parsing algorithm, a chart is filled from left to right and the procedure is expectation-driven; i.e., we know that the start symbol is the parsing goal, and using the rules we direct our search in that direction. In the parsing process, we use the *dotted rules* (as in the Earley's algorithm [6]), e.g.:

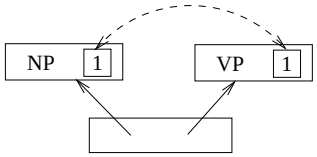
$$\mathrm{NP} \rightarrow \mathrm{THE} \cdot \mathrm{ADJ\ N}.$$

There are several issues that have to be handled differently in the HPSG chart parsing than it is typically done:

Space is not reused. Time is reused, but we do not reuse the space in the HPSG chart parsing. Let us consider the following HPSG rule:

$$\mathsf{S} \rightarrow \left[\begin{array}{l} \mathit{phrase} \\ \mathsf{HEAD}\colon \left[\begin{array}{l} \mathit{noun} \\ \mathsf{AGR}\colon \boxed{1} \end{array}\right] \end{array}\right] \left[\begin{array}{l} \mathit{phrase} \\ \mathsf{HEAD}\colon \left[\begin{array}{l} \mathit{verb} \\ \mathsf{AGR}\colon \boxed{1} \end{array}\right] \end{array}\right]$$

If we apply this rule to two entries, we can represent the result in the following way:



Obviously, entries NP and VP are not independent any more, and they cannot freely participate in other parent constituents. For this reason, we cannot include daughters in their mother simply by referring to them, we have to clone the daughters and then include them in the new mother entry. Fortunately, we do not have to include complete daughter components in a mother entry; instead, we propagate only some information.

Island parsing instead of left-to-right expectation-driven parsing. The island parsing is used instead of the left-to-right expectation-driven parsing. In island parsing, component recognition starts from each word in the sentence and proceeds in both directions. The process could be described in the following way: Each word in the sentence becomes an active entity having a goal of creating connected components according to given rules, in which it will play the head role. For HPSG rules that do not have a head, we can choose a dummy head, the left-most daughter being the best candidate.

The above short description is related to an implementational idea based on creating an active agent for each word in the sentence. It may be questionable whether we can call such entities agents, but in any case they are objects with their own thread of execution and the goal of creating as large as possible syntactic components in which they play the head role. This idea leads to a highly parallelizable parsing algorithm with many relatively independent threads. On a machine with a large number of processors, this might be an efficient approach; however, in our environment the thread management overhead prevails the effect of parallelization, so a serialized version is implemented. The implemented algorithm can easily be turned into a hybrid version as well.

In general, the island parsing is more robust than the left-to-right parsing, and even when a sentence is not fully parsed, the island approach leaves more partial results in the chart to work with. In particular, there are two main advantages of the island parsing when applied to HPSGs:

- efficiency reasons

  The head is typically more restrictive than other components in an HPSG rule. For this reason, it is likely that more options will be pruned right at the beginning if we start to unify daughters from the head.

- natural extension of the HPSG concept

  Starting to parse from the head follows the intuition behind the HPSG formalism. This makes the grammar easier to maintain, and the parser is more likely to behave as a grammar designer would expect it.

After this introductory discussion, we can present the parsing algorithm. We use two charts in the algorithm: a *passive chart,* and an *active chart.* The passive chart contains the completed components, which we call *passive chart entries,* and the active chart contains equivalents of dotted rules—uncompleted components called *active chart entries.* Unlike dotted rules, an active chart entry contains two dots denoting the left and the right bound of the island; i.e., the part that is parsed and instantiated so far. An active chart entry can expand to the left or to the right by instantiating one more daughter and moving the dot in the respective direction, unless it is complete

on that side. If an active entry is complete on the left side, we call it *left-complete*, and if it is complete on the right side, we call it *right-complete.* When the dots reach both ends of the rule, the active chart entry is completed and it is moved to the passive chart. In the above process, we memorize all intermediate active chart entries by applying a version of unify-and-clone operation.

An example of the parsing process is given in Figure 2. The passive chart entries are represented by rounded rectangles, and the active chart entries have arrow-like shapes. An arrow indicates an incomplete side of an active entry.

Passive chart entries and active chart entries are AVMs with some additional bookkeeping data, such as *E.from* and *E.to* fields used to record the span of an entry in the chart. The following algorithm is used to add a passive chart entry to the passive chart:

Algorithm: **AddPassiveEntry(*E*)**

Input: *E* a passive entry

1. add E to the passive chart
2. **For each** rule R in the grammar **do**
3. | Attempt to unify E with the head of R
4. | **If** the unification was successful **then**
5. | | get unified and cloned version of R
6. | | **If** R is complete **then** AddPassiveEntry(R)
7. | | **Else** add R to the active chart

There is not a procedure for adding an active entry (line 7), since we simply add the entry to the chart without any additional operations.

The main parsing algorithm is:

Algorithm: **Parse**

Input: *sentence* the input sentence
Output: *AVMs* a set of AVMs representing complete parses of the sentence

1. **For each** lexeme L of *sentence* **do**
2. | **For each** AVM E corresponding to L in the lexicon **do**

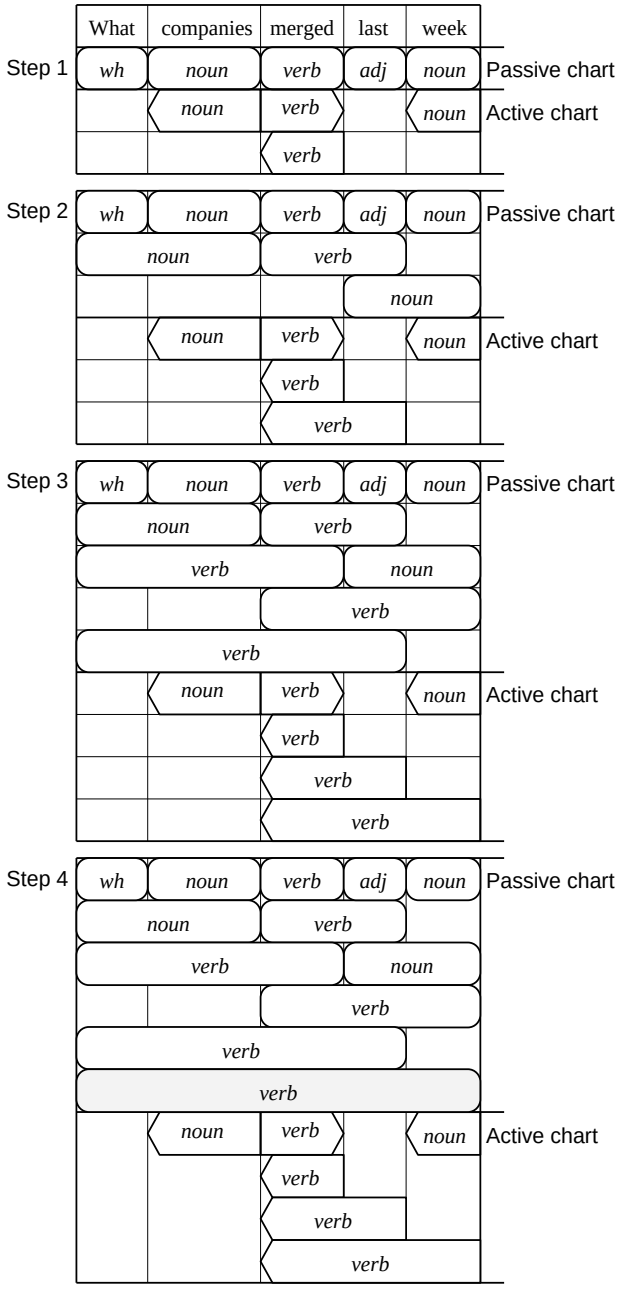


Figure 2: An example of chart parsing

```
3.      |   |AddPassiveEntry(E)
4.  While any entries are added to the charts do
5.      |For each active entry E do
6.      |   |If E is not left-complete then
7.      |   |   |For each passive entry L ending at E.from do
8.      |   |   |   |If NotDone(E, L) then
9.      |   |   |   |   |create new entry E1 by left-expanding E with L
10.     |   |   |   |   |If E1 is successfully created then
11.     |   |   |   |   |   |If E1 is complete then AddPassiveEntry(E1)
12.     |   |   |   |   |   |Else add E1 to the active chart
13.     |   |If E is not right-complete then
14.     |   |   |For each passive entry R starting at E.to do
15.     |   |   |   |If NotDone(E,R) then
16.     |   |   |   |   |create new entry E1 by right-expanding E with R
17.     |   |   |   |   |If E1 is successfully created then
18.     |   |   |   |   |   |If E1 is complete then AddPassiveEntry(E1)
19.     |   |   |   |   |   |Else add E1 to the active chart
20. Return all passive entries spanning the whole sentence
```

The parsing starts by tagging (lines 1–3); i.e., by creating one or more passive chart entries for each recognized lexeme.[2] The entries are extracted from the lexicon. If a lexeme is ambiguous, then all passive entries corresponding to it are added to the chart.

In lines 8 and 15, we use the operation NotDone. This operation is used to keep a record of entry pairs already used in expanding. Only the entries that are not previously used for the operation (i.e., that are "not done") are used.

3 Implementation

In this section, we discuss the important implementational issues.

In Java language, classes are organized into modules called *packages.* There are two packages in the parser Stefy:

`avm` **package:** used to implement AVMs, type lattices, atoms, variables, attributes, and supporting operations including unification; and

[2] A lexeme may consist of several words, so we can have overlapping lexemes.

`hpsg` **package:** used to implement HPSG rules, lexicons, grammars, principles, and the parser.

In the design of code, the emphasis is put on the code efficiency.

We first discuss some general implementation issues, and then a summary descriptions of the two packages are given in the following subsections.

3.1 Grammar development versus parser usage

For most of the classes in these two packages, there are two different ways in which they are used:

1. In the first usage, which we call *debug and develop* (DD), we want to be able to construct grammars and AVMs in a flexible way with a lot of functionality, but without much consideration for neither time nor space efficiency.

2. In the second usage, which we call *run time* (RT), we care about efficiency and we do not need as much functionality as in the previous mode. For example, it is assumed that the lexicon AVMs and the grammar rules are already built, and they are just retrieved and used in the parsing process by applying the operation of unification only.

This division is related to the notions of debug and release software configurations. However, in this case we do not consider the debug and release versions of the packages only, but also the debug and release versions of an NLP grammar that a package user develops.

The code used for RT is also included in the DD part. On the other side, we want the code used exclusively in DD to be clearly separated from RT, so that the RT part can be executed without access to the DD part.

For example, the TypeLattice class belongs to the DD part of the code, while the Type class is in the RT part. Sometimes, we want to have two versions of the same class that separate those two functionalities. As an example, the class Type is RT, while the class TypeX is DD. Adding 'X' in the name of class used for DD is the convention used throughout the code.

It would be natural to separate the DD and RT parts into two separate Java modules. However, it is not done since the classes in DD frequently need additional access rights in the RT part, which are provided only to the code in the same package.

3.2 Package `avm`

The package `avm` implements AVMs in Java, with necessary data structures, including: atoms, type lattices, attributes, and variables. Among other operations, the operation of AVM unification is implemented. This package is separated from the `hpsg` package, since AVMs and unification can be used independently in many applications, other than HPSG parsing.

BasicAVM (RT). The abstract class BasicAVM is the superclass of classes Atom, Variable, and BasicAVMTuple. It is defined according to Definition 2.

Atom (RT). The class Atom is subclass of BasicAVM. It implements an atom. Each atom is associated with a string. The atoms representing equal strings are the same (as pointers), so they are easily and efficiently compared.

AtomSet (RT). The class AtomSet implements a set of atoms. It provides functions for adding atoms to the set, and for retrieving them by name.

AttributeSet (DD). The class AttributeSet implements a set of attributes. It provides methods for managing an attribute set. There is no class Attribute, since attributes are not directly represented in an AVM. When attributes are added to the attribute set, they are assigned an integer key (starting from 0), which is not changed afterwards.

BasicAVMTuple (RT). The class BasicAVMTuple implements a basic AVM tuple. It contains the tuple type, and an array of variables. The attributes are not represented explicitly for efficiency reasons. Each attribute is represented by its key, which is an index of the value of that attribute in the variable array (after possibly adding an offset).

The actual implementation is somewhat more efficient than a direct approach. A part of future work is finding out experimentally if an implementation with sorted lists would be more efficient.

For DD operations, we use the class BasicAVMTupleX.

BasicAVMTupleX (DD). This class is used to manipulate basic AVM tuples in the DD phase. The operations include creating AVM tuples, and setting types by type name.

Type (RT). The class Type implements an instance of type. It is implemented as a bit array (by extending the standard `java.util.BitSet` class). The type *bot* is represented by the set of zero-bits (false values), while the type *top* is represented by the set of one-bits (true values). All types are of the same bit size. Each non-*bot* type is associated with one bit, and the bit set associated with a type has true values for bits corresponding to all its subtypes (except *bot*). Therefore, any two types are unified by doing bitwise operation AND on their bit sets.

This implementation relies on the claim

$$\{t \in T : t \leq t_1 \wedge t_2\} = \{t \in T : t \leq t_1\} \cap \{t \in T : t \leq t_2\},$$

for any two types $t_1, t_2 \in T$. If $t \leq t_1 \wedge t_2$, then $t \leq t_1$ and $t \leq t_2$ since $t_1 \wedge t_2 \leq t_1$ and $t_2 \wedge t_2 \leq t_2$. In the other direction, if $t \leq t_1$ and $t \leq t_2$, then $t \leq t_1 \wedge t_2$ by the definition of the operation $\wedge$ (Definition 1). This proves the above claim.

Since it is used only in RT, the class Type does not contain information about type names. For mapping between type names and their representations, we use TypeLattice.

TypeX (DD). The class TypeX is used within TypeLattice. A type lattice is built in the DD phase. All functionality needed in this phase, local to an individual type, and not needed in the RT phase is implemented in this class. This functionality includes: finding the type name and retrieving information about direct parents of a type.

TypeLattice (DD). The class TypeLattice implements a lattice of types. It provides operations for defining new types, finding type names, building the type lattice, etc.

A type lattice has to be completed before we can to use it to create AVMs. When we add new types, we do not maintain all necessary information needed for the lattice of types. We only maintain type names, the total number of types, and some subtype relations. We do not maintain all subtype relations,

but all of them can be obtained by a transitive closure of the existing ones. In order to bring a TypeLattice instance to a consistent state, the algorithm Normalize is applied. The preconditions of the algorithm Normalize are:

1. all types are included and assigned some id numbers, and

2. the reflexive and transitive closure of all included subtype relations is a partial order (there are no cycles).

The algorithm post-conditions are:

1. only direct subtype links (which cannot be inferred by transitive closure) remain memorized,

2. bit sets are properly set, i.e., the bit set of each type has to indicate all types that are subtypes of this type, and

3. the type id numbers have to be assigned in a topological-sort order except for the type *bot*; more precisely, $bot.id = -1$, $top.id = 0$, and for all other types t_1 and t_2, t_1 supertype of t_2 implies $t_1.id < t_2.id$. The number $t.id$ denotes the index of the bit in a bit set corresponding to the type t.

The following algorithm is used:

Algorithm: **Normalize**(T)

Input:	T	type lattice satisfying preconditions
Output:	T	normalized type lattice (i.e., satisfying the post-conditions

1. do topological sort of the lattice (*top* is the first element)
2. remove *bot* and assign $bot.id \leftarrow -1$
3. assign id numbers in order, set appropriate bits in bit sets, and set bits resulting from stored subtype relations,
4. do transitive closure using an algorithm based on Floyd-Warshall algorithm ([5] p.563) and remove subtype relations which are not direct.

Variable (RT). The class Variable extends the class BasicAVM. Its basic function is to be a reference to another BasicAVM through the field p. Variables are used to form UNION-FIND structures (e.g., see [5]) that represent disjoint sets of unified variables. The representative of a set is a variable, whose value of p may be `null`, or a basic AVM that is not a variable.

Beside p, a variable contains some other auxiliary fields: *rank*, *newp*, *newrank*, *status*, and *next*; and some static fields. The field *rank* is used for efficient operations on the UNION-FIND structures. The usage of other fields is described below.

There are various ways how two AVMs can be unified. When we attempt to unify two AVMs, a question is: What happens when unification fails? One expensive solution to the problem is to clone both AVMs, and attempt to unify the clones. If the unification succeeds, return the unified clone, otherwise fail and keep the original AVMs. Another cheaper approach, which is used here, is to record any changes; and, if unification fails, recover the old state, otherwise commit the changes. Additionally, this approach is efficiently used in an important kind of unification—the `unifyAndClone` method.

During unification, we change values of *newp* and *newrank*, instead of p and *rank*. Additionally, each changed variable is added to the static list of changed variables using the field *next* for linking. After failed or successful unification, it is now easy to recover or commit the variable states: we simply go through the list of changed variables and update fields.

In the unify-and-clone operation, or non-destructive unification, we want both results, the old and the new structure, after unification. The first phase is attempting to unify two AVMs. If the unification fails, we recover the old state and the procedure is finished. If the unification succeeds, we perform two additional operations: *duplicate* and *separate*.

In the operation duplicate, we make a depth-first search of the AVM graph, and for each node we create a shallow clone.

In the operation separate, a new depth-first search is done, in which we separate the old and the new structure.

Both depth-first searches run in linear time, so the total unification time remains to be $O(n \cdot \alpha(n))$. The field *status* is used for the depth-first searches (this field is sometimes called *color*).

VariableX (DD). The VariableX class is used to separate DD functionality from the class Variable. It provides methods for building AVMs, printing AVMs, etc.

DD (DD). DD is a generic class used for various DD tasks. It imports and frequently uses the standard `java.lang.reflect` package.

ErrorHandler (RT). The class used to handle errors and exceptions during run time. The only type of errors that currently exists are fatal errors.

ErrorHandlerX (DD). This class is similar to ErrorHandler, except that it prints out more details about an error. The class ErrorHandlerX extends the class ErrorHandler.

3.3 Package `hpsg`

The package `hpsg` uses package `avm` and, on top of it, defines HPSG rules, charts, grammars, and lexicons. The parsing algorithm is implemented as a Grammar method.

Rule (RT). The class Rule implements an HPSG rule. It is a subclass of the class `avm.Variable`, and it also implements the interface Registerable, which is described below.

A rule is a variable that describes the left-hand side of an HPSG rule. Additionally, a rule contains an array of variables representing rule daughters, and the head index field.

For efficiency and simplicity reasons, rules are used as passive and active entries in the charts, so additional fields for storing the entry and island boundaries are defined.

RuleX (DD). Similarly to VariableX, the RuleX class contains functionality used for creating and maintaining the Rule instances, which is not needed at run-time. The superclass of RuleX is VariableX.

Registerable interface (RT). The interface Registerable defines only one method prototype: `id()`, which returns a String identifier that uniquely identifies an object.

Register (RT). The class Register and the interface Registerable are used to implement the method NotDone described in the parsing algorithm.

RegisterX (DD). The RegisterX class separates exclusive DD functionality from the Register class.

Chart (RT). Both charts used in the parsing algorithm, the passive and active chart, are instances of the class Chart. It provides method for efficiently adding and accessing chart entries; i.e., instances of Rule.

ChartX (DD). The ChartX class separates exclusive DD functionality from the Chart class.

Lexicon (RT). The Lexicon class provides a generic lexicon class. New lexicons can be defined by extending this class.

Grammar (RT). Similarly to Lexicon, the Grammar class provides a generic grammar class. A package user can provide new grammars by extending this class.

4 Discussion

The HPSG formalism is usually based on the theory of typed feature structures described in Carpenter 1992 [1]. The definition presented here provides for an easy implementation in programming languages that do not include a built-in unification mechanism. Additionally, we also provide an explicit and efficient unification algorithm for AVMs. The presented formalism is very compact and concise.

In the second part of the paper, we describe an implementation of Java parser for HPSGs based on the formalism presented in the first part.

A drawback of this implementation is that it is not suitable for development of the grammar and lexical resources. Other systems, like ALE [3] and LKB [4], are more appropriate for this task. After a grammar or a lexicon is developed in one of those systems, it is translated into a Java description and used in our system.

5 Conclusion

In this paper, we have presented a new precise and compact description of the HPSG formalism, which is very suitable for implementation of HPSG parsers in low-level languages.

It is proven that the grammar defines the set of recursively enumerable languages.

A Java parser for HPSG, called Stefy, is implemented using this formalism. It represents an important step towards applying HPSG formalism in the area of *distributed NLP* and *answer extraction.* A detailed documentation of the implemented parser is included.

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] Bob Carpenter. Probabilistic word graph parser: Java source & documentation, 1999. `http://www.colloquial.com/carp/Parser/`.

[3] Bob Carpenter and Gerald Penn. ALE, the attribute logic engine, user's guide. WWW, May 1999.
`http://www.sfs.nphil.uni-tuebingen.de/~gpenn/ale.html`.

[4] Ann Copestake. *The (new) LKB system, Version 5.2*, October 1999.
`http://www-csli.stanford.edu/~aac/lkb.html` .

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1989.

[6] J. Earley. An efficient context-free parsing algorithm. *Communications of the ACM*, 13(2):94–102, February 1970.

[7] John E. Hopcroft and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Addison-Wesley, 1979.

[8] G. Huet. *Résolution d'equations dans les langages d'ordre 1,2, ...,ω.* Thèse de Doctorat d'Etat, Université de Paris 7 (France), 1976.

[9] Vlado Kešelj. Java parser for HPSGs: Why and how. In Nick Cercone, Kiyoshi Kogure, and Kanlaya Naruedomkul, editors, *Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING'99*, pages 288–294, Waterloo, Ontario, Canada, August 1999. `http://www.cs.uwaterloo.ca/~vkeselj/papers/pacling99.http`.

[10] Paul King. An expanded logical formalism for Head-Driven Phrase Structure Grammar. Arbeitspapiere des sfb 340, University of Tübingen, 1994.

[11] Kevin Knight. Unification: a multidisciplinary survey. *ACM computing surveys, March 1989*, 21(1):93–124, 1989.

[12] Takaki Makino, Minory Yoshida, Kentaro Torisawa, and Jun'ichi Tsujii. LiLFeS—towards a practical HPSG parser. In *Proceedings of the COLING-ACL 98, Montreal*, pages 807–811, 1998.

[13] Rob Malouf and John Carroll. Efficient graph unification for parsing constraint-based grammars. Paper presented at the Computer Science Department, Duke University, 1999.
`http://hpsg.stanford.edu/rob/papers/duke-slides.ps.gz`.

[14] Carl J. Pollard and Ivan A. Sag. *Head-Driven Phrase Structure Grammar.* University of Chicago Press, Chicago, 1994.

[15] Fred Popowich and Carl Vogel. Chart parsing head-driven phrase structure grammar. Technical Report CSS-IS TR 90-01, Simon Fraser University, 1990.

[16] Derek Proudian and Carl J. Pollard. Parsing head-driven phrase structure grammar. In *Proceedings of the Twenty-Third Annual Meeting of the ACL*, pages 167–171, Chicago, IL, 1985. ACL.

[17] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[18] Ivan A. Sag and Thomas Wasow. *Syntactic Theory: A Formal Introduction*. CSLI Publications, Stanford, 1999.