Computing Phrasal-signs in HPSG prior to Parsing

Kentaro Torisawa and **Jun'ichi Tsujii**
Department of Information Science, University of Tokyo,
Hongo 7-3-1, Bunkyo-ku, Tokyo, 113, Japan
{torisawa,tsujii}@is.s.u-tokyo.ac.jp

Abstract

This paper describes techniques to compile lexical entries in HPSG (Pollard and Sag, 1987; Pollard and Sag, 1993)-style grammar into a set of finite state automata. The states in automata are possible signs derived from lexical entries and contain information *raised* from the lexical entries. The automata are augmented with feature structures used by a *partial unification routine* and *delayed/frozen* definite clause programs.

1 Introduction

Our aim is to build an efficient and robust HPSG-based parser. HPSG has been regarded as a sophisticated but fragile and inefficient framework. However, its *principle-based* architecture enables a parser to handle real world texts only by giving concise *core* grammar, including principles and templates for lexical entries, **default lexical entries**(Horiguchi et al., 1995). The architecture is different from those of conventional unification-based formalisms which require hundreds of CFG skeletons to parse real world texts.

However, these design principles of HPSG have draw-backs in parsing cost. That is, signs/feature structures corresponding to non-terminal symbols in CFG become *visible* only after applying principles and a parser has to create feature structures one by one using unification. In addition, identity checking of non-terminal symbols used to eliminate spurious signs must be replaced with subsumption checking, which further deteriorates efficiency.

Our grammar compiler computes skeletal part of possible phrasal-signs from individual lexical entries prior to parsing, and generates a set of finite state automata from lexical entries to avoid the above draw-backs. We call this operation **Off-line raising** and an automaton thus generated is called a **Lexical Entry Automaton (LA)**. Its states corresponds to part of signs and each transition between states corresponds to application of a **rule schema**, which is a non-lexical component of grammar.

Our parsing algorithm adopts a two-phased parsing method.

Phase 1 Bottom-up chart-like parsing with LAs.

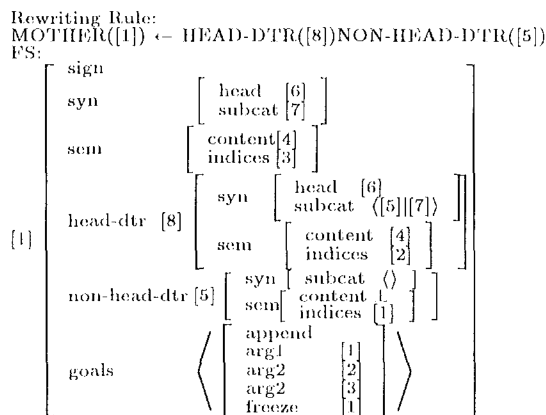


Figure 1: An example of a rule schema.

Phase 2 Computing part of feature structures which cannot be computed at compile-time.

We call the feature structures that are represented as states in automata and are computed at compile-time **Core-structures**, and the feature structures which are to be computed in Phase 2, **Sub-structures**. In Phase 1 parsing, a core-structure correspond to a state in an LA. The cost of computing sub-structures at Phase 2 is minimized by **Dependency Analysis** and **Partial Unification**.

The next section describes rule schemata, central components of the formalism, and gives a definition of Definite Clause Programs. Section 3 describes how to obtain LAs from lexical entries and how to perform the Phase 1 parsing. Section 4 explains the Phase 2 Parsing algorithm. A parsing example is presented in Section 5. The effectiveness of our method is exemplified with a series of experiments in Section 6.

2 Rule Schemata and Definite Clause Programs

Our formalism has only one type of component as non-lexical components of grammar, i.e., **rule schemata**.[1] An example is shown in Figure 1. A rule schema consists of the following two items.

[1] In our current system, rule schemata are generated from principles and rewriting rules according to a specification given by a programmer.

rule(R) a rewriting rule without specific syntactic categories;

fs(R) a feature structure.

A characteristic of HPSG is in the flexibility of principles which demands complex operations, such as *append* or *subtraction* of list-value feature structures. In our formalism, those operations are treated by a Definite Clause Program. A DCP can be seen as a logic program language whose arguments are feature structures. An *auxiliary term*, a query to a DCP augmenting a rule schema, is *embedded* in a feature structure of a rule schema as the value of `goals`. The rule schema in the example has an auxiliary term, *append*([1], [2], [3]).

The bottom-up application of the rule schema R is carried out as follows. First, two daughter signs are *substituted* to the `HEAD-DTR` position and `NON-HEAD-DTR` position of the rewriting rule $rule(R)$. Then, the signs are unified with the `head-dtr` value and the `non-head-dtr` value of the feature structure of the schema, $fs(R)$. Finally, the auxiliary term for DCPs given in the schema is evaluated.

Our definition of a DCP has a more operational flavor than that given by Carpenter(Carpenter, 1992). The definition is crucial to capture the correctness of our method.[2]

Definition 1 (DCP) *A definite clause program (DCP) is a finite set of feature structures, each of which has the following form.*

$$\left[\begin{array}{ll}\texttt{goals} & \langle H | [1] \rangle \\ \texttt{next-steps} & [\ \texttt{goals}\ \langle B_0, B_1, \cdots, B_n, [1] \rangle\ ]\end{array}\right]$$

[3] *where* $0 \leq n$ *and* $H, B_0, \cdots, B_n$ *are feature structures.*

A feature structure of the above form corresponds to a clause in Prolog. $H, B_0, \cdots, B_n$ corresponds to literals in Prolog. H is the head and $B_0, \cdots, B_n$ are literals in the body of a clause.

Definition 2 (Execution of DCP) *Execution of a DCP P for the query,*

$$Query = [\ \texttt{goals}\ \langle q_0, q_1, \cdots q_l \rangle\ ]$$

is a sequence of unification,

$$Query \sqcup r_1 \sqcup r_2 \sqcup \cdots \sqcup r_n$$

where $r_i = [\ (\texttt{next-steps})^{i-1}\ \ C_i\ ]$, $C_i \in P$ *or* $C_i = [\ \texttt{goals}\ \langle\rangle\ ]$. *If the execution is terminated,* C_n *must be unifiable with* $[\ \texttt{goals}\ \langle\rangle\ ]$. *In this case, we call the sequence* $\langle r_1, \cdots, r_n \rangle$ *a* **resolution sequence**.

[2]Though, through the rest of the paper, we treat the definition as if it were used in an actual implementation, the actual implementation uses a more efficient method whose output is equivalent with the result obtained by the definition.

[3] $\langle B_0, \cdots, B_n, [1] \rangle$ is an abbreviation of $\left[\begin{array}{ll}\texttt{first} & B_0 \\ \texttt{rest} & \left[\cdots \left[\begin{array}{ll}\texttt{first} & B_n \\ \texttt{rest} & [1]\end{array}\right]\right]\end{array}\right]$

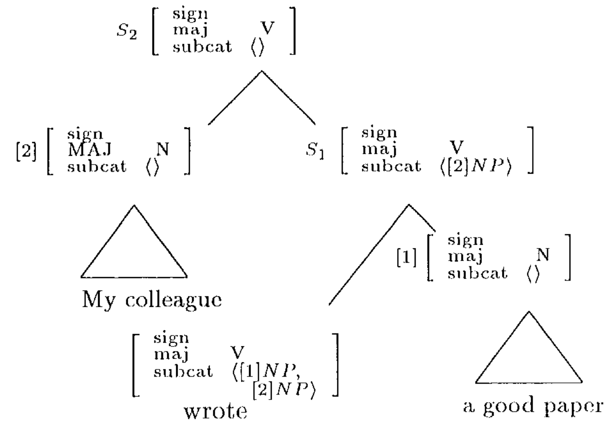


Figure 3: A parsing example

$(\texttt{next-steps})^{i-1}\ |\texttt{goals}$ of $Query \sqcup r_1 \sqcup r_2 \sqcup \cdots \sqcup r_i$ represents the goals which are to be solved in the steps following the i-th *step*. The goals are *instantiated* by the steps from the first one to i-th one, through structure sharings. The result of execution in a Prolog-like sense appears in the query. Figure 2 is an example of execution for the query `append([a],[b],X)` , whose definition is based on a standard definition of `append` in Prolog.

Given this definition of DCPs, an application of a rule schema to two daughter signs D_1 and D_2 can be expressed in the following form, where $\langle r_1, r_2, \cdots, r_n \rangle$ is a resolution sequence:

$$M = \left[\begin{array}{ll}\texttt{head-dtr} & D_1 \\ \texttt{non-head-dtr} & D_2\end{array}\right] \sqcup fs(R) \sqcup r_1 \sqcup r_2 \sqcup \cdots \sqcup r_n$$

3 Lexical Entry Automata

This section presents a **Lexical Entry Automaton (LA)**. The inefficiency of parsing in HPSG is due to the fact that what kind of constituents phrasal-signs would become is *invisible* until the whole sequence of applications of rule schemata is completed. Consider the parse tree in Figure 3. The phrasal-signs S_1 and S_2 are invisible until a parser creates the feature structures describing them, using expensive unification.

Our parsing method avoids this on-line construction of phrasal-signs by computing skeletal part of parse trees prior to parsing. In Figure 3, our compiler generates S_1 and S_2 only from the lexical entry "wrote," without specifying the non-head daughters indicated by the triangles in Figure 3. Since the non-head daughters are token-identical with `subcat` values of the lexical entry for "wrote", the obtained skeletal parse tree contains the information that S_1 takes a noun phrase as object and S_2 selects another noun-phrase. Then unifying those non-head daughters with actual signs constructed from input, parsing can be done. An LA expresses a set of such skeletal parse trees. A state in an LA corresponds to a phrasal-sign such as S_1 and S_2. They are called **core-structures**. A transition arc is a domination link between a phrasal-sign and its head daughter, and its condition for transition on input is a non-head

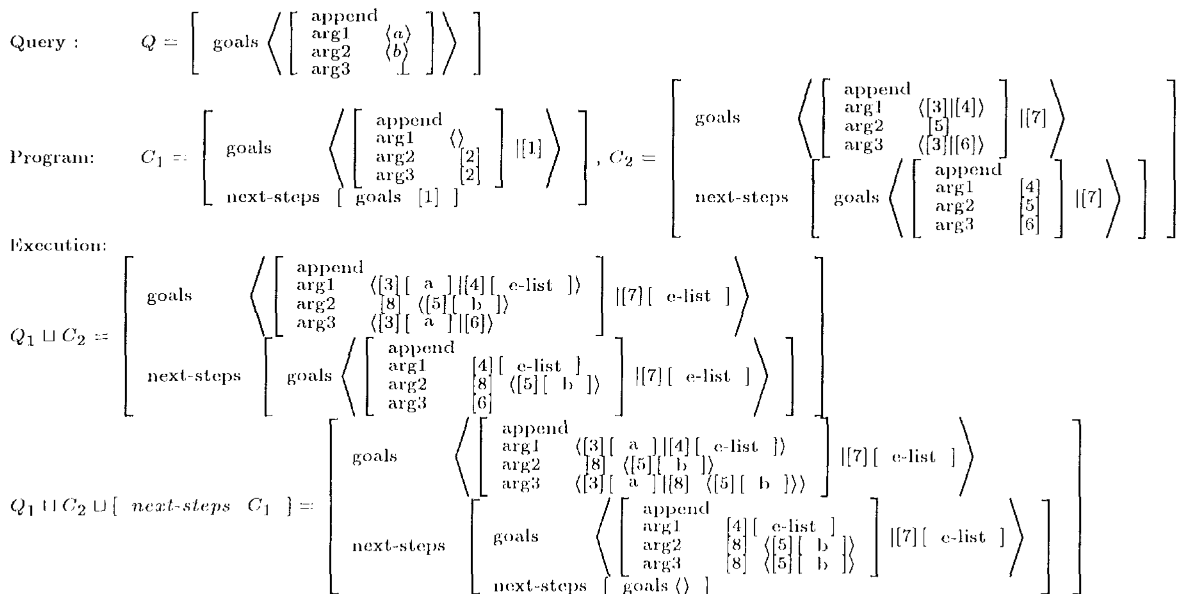


Figure 2: An example of DCP's execution

daughter, such as signs tagged [1] and [2] in Figure 3. Kasper *et al.* presented an idea similar to this *off-line raising* in their work on HPSG-TAG compiler(Kasper et al., 1995). The difference is that our algorithm is based on substitution, not adjoining. Furthermore, it is not clear in their work how off-line raising is used to improve efficiency of parsing.

Before giving the definition of LAs, we define the notion of a quasi-sign, which is *part* of a sign and constitutes LAs.

Definition 3 (quasi-sign(n)) *For a given integer n, a feature structure S is a quasi-sign(n) if it has some of the following four attributes:* `syn, sem, head-dtr,non-head-dtr` *and does not have values for the paths* (`head-dtr` + `non-head-dtr`)^n.

A *quasi-sign*(n) cannot represent a parse tree whose height is more than n, while a sign can express a parse tree with any height. Through the rest of this paper, we often extract a *quasi-sign*(n) S from a sign or a *quasi-sign*(n') S' where $n \leq n'$. This operation is denoted by $S = ex(S', n)$. This means that S is equivalent to S' except for the attributes `head-dtr` and `non-head-dtr` whose root is the (`head-dtr` + `non-head-dtr`)^n value in S'. Note that S and S' are completely different entities. In other words, S and S' pose different scopes on structure sharing tags. In addition, we also extract a feature structure F reached by a path or an attribute p in a feature structure F'. We denote this by $F = val(F', p)$ and regard F and F' as different entities.

Definition 4 (Lexical Entry Automaton(LA)) *A Lexical Entry Automaton is a tuple $\langle Q, A, q_0 \rangle$ where,*

Q : *a set of states, where a state is a quasi-sign*(0).

A : *a set of transition arcs between states, where a transition arc is a tuple $\langle q_d, q_m, N, D, R \rangle$ where $q_d, q_m \in Q$, N is a quasi-sign*(0), *D is a quasi-sign*(1) *and R is a rule schema.*

q_0 : *the initial state, which corresponds to a lexical entry.*

In a transition arc $\langle q_d, q_m, N, D, R \rangle$, q_m denotes the destination of the transition arc, and q_d is the root of the arc. The N is a non-head daughter of a phrasal-sign, i.e., the destination state of the transition, and expresses the input condition for the transition. The D is used to represent the dependency between the mother sign and the daughters through structure sharings. This is called a **Dependency Feature Structure(DFS)** of the transition arc, the role of which will be discussed in Section 4. R is the rule schema used to create this arc.

An LA is generated from a lexical entry l by the following recursive procedure:

1. Let $\mathcal{S}$ be $\{l\}$, $\mathcal{A}$ be an empty set and $s_d = l$
2. For each rule schema R, and for each of its each resolution sequence $\langle r_1, \cdots, r_n \rangle$ obtain,

$$D = [\ \texttt{head-dtr}\ s_d\ ] \sqcup fs(R) \sqcup r_1 \sqcup \cdots \sqcup r_n$$

   and if D is a feature structure, obtain $s_m = ex(D, 0)$ and $N = ex(val(D, \texttt{non-head-dtr}), 0)$.
3. If D is a feature structure,
   - If there is a state $s'_m \in \mathcal{S}$ such that $s'_m \approx s_m$,[4] let s_m be s'_m. Otherwise, add s_m to $\mathcal{S}$.
   - If there is no $Tr = \langle s''_d, s''_m, N'', D'', R \rangle \in \mathcal{A}$ such that $s_m \approx s''_m$, $s_d \approx s''_d$, $N \approx$

[4]For any feature structures f and f', $f \approx f'$ iff $f \sqsubseteq f'$ and $f' \sqsubseteq f$

```
Phase2-proc-dep(e : edge);
assume e = ⟨l, r, S, Dep⟩
return S ⊔ sub-structure(e)

sub-structure(e : edge);
assume e = ⟨l, r, S, Dep⟩
If Dep = φ
then return sub(S),
else
  for each ⟨D, e_h, e_n, R⟩ ∈ Dep,
  assume that e_h = ⟨l_h, r_h, S_h, Dep_h⟩
    and e_n = ⟨l_n, r_n, S_n, Dep_n⟩
    S_h := sub-structure(e_h),
    S_n := S_n ⊔ sub-structure(e_n)
    If neither of S_h and S_n is nil,
      sub_0 :=
        p_unify(dep(D) ⊔ sub(fs(R)),
          [ head-dtr       S_h ]
          [ non-head-dtr   S_n ] ,
        rs)
        ..............................(A)
    for each resolution sequence
      ⟨r_1, ··· , r_i⟩,
        sub := sub_0 ⊔ r_1 ⊔ ··· ⊔ r_i
      ..............................(B)
    If sub is not a feature structure or
      either of S_h or S_n is nil,
    then return nil
    else return sub
```

Figure 4: A recursive procedure for the Phase 2

N'' and $D \approx D''$, then, add the tuple $\langle s_d, s_m, N, D, R\rangle$ to $\mathcal{A}$.

4. If the new *quasi-sign*(0) (s_m) was added to S in the previous step, let s_d be s_m and go to Step 2.

When this terminates, $\langle \mathcal{S}, \mathcal{A}, l\rangle$ is the LA for l.

The major difference of Step 2 and the normal application of a rule schema is that non-head-dtr values are not specified in Step 2. In spite of this underspecification, certain parts of the non-head-dtr are instantiated because they are token-identical with certain values of the head-dtr domain. By unifying non-head-dtr values with *actual* signs to be constructed from input sentences, a parser can obtain parsing results. For more intuitive explanation, see (Torisawa and Tsujii, 1996).

However, this simple LA generation algorithm has a termination problem. There are two potential causes of non-termination. The first is the generative capacity of a feature structure of a rule schema, i.e., a rule schema can generate infinite variety of signs. The second is non-termination of the execution of DCP in Step 2 because of lack of *concrete* non-head daughters.

For the first case, consider a rule schema with the following feature structure.

$$\begin{bmatrix} \texttt{syn} & \begin{bmatrix} \texttt{counter} \ \langle bar | \texttt{[1]} \rangle \end{bmatrix} \\ \texttt{head-dtr} & \begin{bmatrix} \texttt{syn} & \begin{bmatrix} \texttt{counter} & \texttt{[1]} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Then, this can generate an infinite sequence of signs, each of which contains a part, [counter $\langle bar, bar, \cdots, bar\rangle$] and is not equivalent to any previously generated sign. In order to resolve this difficulty, we apply the **restriction** (Shieber, 1985) to a rule schemata and a lexical entry, and split the feature structure $F = fs(R)$ of a rule schema R or a lexical entry $F = l$, into two, namely, $core(F)$ and $sub(F)$ such that $F = core(F) \sqcup sub(F)$. The definition of the restriction here is given as follows.

Definition 5 (paths) *For any node n in a feature structure F, paths(n, F) is a set of all the paths that reaches n from the root of F.*

Definition 6 (Restriction Schema) *A restriction schema rs is a set of paths.*

Definition 7 (Res) *$F' = Res(F, rs)$ is a maximal feature structure such that each node n in F' satisfies the following conditions.*

- *There is a node n_o in F such that $paths(n_o, F) = paths(n, F')$ and $type(n) = type(n_o)$.*
- *For any $p \in paths(n, F')$, there is no path $p_r \in rs$ which prefixes p.*

Res eliminates the feature structure nodes which is specified by a restriction schema. For a certain given restriction schema rs, $core(fs(R)) = Res(fs(R), rs)$ and $sub(fs(R))$ is a minimal feature structure such that $core(fs(R)) \sqcup sub(fs(R)) = fs(R)$. The nodes eliminated by *Res* must appear in $sub(fs(R))$. In the example, if we add ⟨syn, counter⟩ to a restriction schema and replace $fs(R)$ with $core(fs(R))$ in the algorithm for generating LAs, the termination problem does not occur because LAs can contain a loop and equivalent signs are reduced to one state in LAs. The $sub(fs(R))$ contains the syn|counter, and the value is treated at Phase 2.

The other problem, i.e., termination of DCPs, often occurs because of underspecification of the non-head-dtr values. Consider the rule schema in Figure 1. The *append* does not terminate at Phase 2 because the indices value of non-head daughters is [⊥]. (Consider the case of executing $append(X, \langle b\rangle, Y)$ in Prolog.) We introduce the *freeze* functor in Prolog which delays the evaluation of the second argument of the functors if the first argument is not instantiated. For instance, freeze(X,append(X,[b],Z)) means to delay the evaluation of *append* until X is instantiated. We introduce the functor in the following form.

$$\begin{bmatrix} \text{goals} & \left\langle \begin{bmatrix} \text{append} & \\ \text{arg1} & \text{[1]} \\ \text{arg2} & \langle b\rangle \\ \text{arg3} & \text{[2]} \\ \text{freeze} & \text{[1]} \end{bmatrix} \right\rangle \end{bmatrix}$$

This means the resolution of this query is not performed if [1] is [⊥]. The delayed evaluation is considered later when the non-head-dtr values are instantiated by an *actual* sign. Note that this change does not affect the discussion on the correctness of our parsing method, because the difference can be seen as only changes of order of unification.

Now, the two phases of our parsing algorithm can be described in more detail.

Phase 1 : Enumerate possible *parses* or edges in a chart only with unifiability checking in a bottom-up chart-parsing like manner.

Phase 2 : For completed parse trees, compute sub-structures by DFSs, $sub(fs(R))$ for each schema R and *frozen* DCP programs.

Note that, in Phase 1, unification is replaced with unifiability checking, which is more efficient than unification in terms of space and time. The *intended* side effect by unification, such as building up logical forms in `sem` values, is computed at Phase 2 only for the parse trees covering the whole input.

3.1 Phase 1 Parsing

The Phase 1 parsing algorithm is quite similar to a bottom-up chart parsing for CFG. The algorithm has a chart and edges.

Definition 8 (edge) *An edge is a tuple* $\langle l, r, S, Dep\rangle$ *where,*

- l *and* r *are vertexes in the chart.*
- S *is a state of an LA.*
- Dep *is a set of tuples in the form of* $\langle D, e_h, e_n, R\rangle$ *where* e_h *and* e_n *are edges,* D *is a* $quasi\text{-}sign(1)$ *and* R *is a rule schema.*

The intuition behind this definition is,

- S plays the role of a non-/terminal in CFG, though it is actually a $quasi\text{-}sign(0)$.
- e_h and e_n denote a head daughter *edge* and a non-head daughter *edge*, respectively.
- *Dep* represents the dependency of an edge and its daughter edges. Where $\langle D, e_h, e_n, R\rangle \in Dep$, D is a DFS of a transition arc. Basically, Phase 1 parsing creates these tuples, and Phase 2 parsing uses them.

The Phase 1 parsing consists of the following steps. Assume that a word in input has a lexical entry L_i and that an LA $\langle Q_i, A_i, q_0^i\rangle$ generated from L_i is attached to the word:

1. Create an edge $l_i = \langle j_i, j_i + 1, q_0^i, \phi\rangle$ in the chart for each L_i, for appropriate j_i.
2. For an edge e_1 whose state is q_1 in the chart, pick up an edge e_2 which is adjacent to e_1 and whose state is q_2.
3. For a transition arc $\langle q_1, q, N, D, R\rangle$, check if N is unifiable with q_2.
4. If the unifiability check is successful, find an edge $d = \langle m_d, n_d, q, Dep_d\rangle$ strictly covering e_1 and e_2.
5. if there is, replace d with a new edge $\langle m_d, n_d, q, Dep_d \cup \{\langle D, e_1, e_2, R\rangle\}\rangle$ in the chart.
6. Otherwise, create a new edge $\langle m, n, q, \{\langle D, e_1, e_2, R\rangle\}\rangle$ strictly covering e_1 and e_2.
7. Go to step 2.

4 Phase 2 Parsing

The algorithm of Phase 2 parsing is given in Figure 4. The procedure *sub-structure* is a recursive procedure which takes an edge as input and builds up sub-structures, which is *differential* feature structures representing *modifications* to core-structures, in a bottom-up manner. The obtained sub-structures are unified with core-structures when 1) the input edge covers a whole input or 2) the edge is a non-head daughter edge of some other edge. Note that the *sub-structure* treats $sub(fs(R))$, a feature structure eliminated by the restriction in the generation of LAs, (the (A) part in Figure 4) and *frozen* goals of DCPs, by additional evaluation of DCPs. (the (B) part)

Here, we use two techniques: One is dependency analysis which is embodied by the function *dep* in Figure 4. The other is a partial unification routine expressed by p_unify in the figure.

The dependency analysis is represented with the function, $dep(F, rs)$, where F is a DFS and rs is a restriction schema used in generation of LAs:

Definition 9 (dep) *For a feature structure* F' *and the restriction schema* rs, $F = dep(F', rs)$ *is a maximal feature structure such that any node* n *in* F *satisfies the conjunction of the following two conditions:*

1. *There is a node* n' *in* F' *such that* $paths(n, F) = paths(n', F')$ *and* $type(n) = type(n')$.
2. *Where A)* $n_d = n$ *or B)* n_d *is a descendant*[5] *of* n, $paths(n_d, F)$ *contains a path prefixed by one of* ⟨`head-dtr`⟩, ⟨`non-head-dtr`⟩ *and* ⟨`goals`⟩.
3. *The disjunction of the following three conditions is satisfied where A)* $n_d = n$ *or B)* n_d *is a descendant of* n.
   - *For some* $p \in paths(n_d, F)$, *there is a path* $p_r \in rs$ *which prefixes* p.
   - *Some* $p \in paths(n_d, F)$ *is prefixed by* $\langle goals\rangle$.
   - *There is no node* n_a *in* F *such that i) there is paths* $p_1, p_2 \in paths(n_a, F)$ *such that* p_1 *is prefixed by* ⟨`syn`⟩ *or* ⟨`sem`⟩ *and* p_2 *is prefixed by* ⟨`head-dtr`⟩ *or* ⟨`non-head-dtr`⟩, *and ii) for any* $p \in paths(n_d, F)$ *there is* $p_a \in paths(n_a, F)$ *which prefixes* p.

Roughly, *dep* eliminates 1) the descendant nodes of the node which appears both in `syn`/`sem` domains and `head-dtr`/`non-head-dtr` domains and 2) the nodes appearing only in `syn`/`sem` domains, except for the node which appears in $sub(fs(R))$ or `goals` domains. In other words, it removes the feature structures that have been already raised to core-structures or other DFSs, except for the structure sharings, and leaves those which will be *required* by DCPs or $sub(fs(R))$.

$p_unify(F_1, F_2, rs)$ is a *partial unification routine* where F_1 and F_2 are feature structures, and rs is a restriction schema used in generation of LAs. Roughly, it performs unification of F_1 and F_2 only for *common* part of F_1, F_2, and it produces unified results only for the node n in F_1 if

[5] n_1 is a descendant of n_2 in a feature structure F iff $n_1 \neq n_2$, and there are paths $p_1 \in paths(n_1, F)$ and $p_2 \in paths(n_2, F)$, and p_2 prefixes p_1.

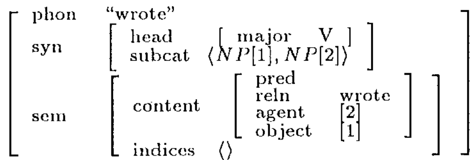


Figure 5: A lexical entry for "wrote"

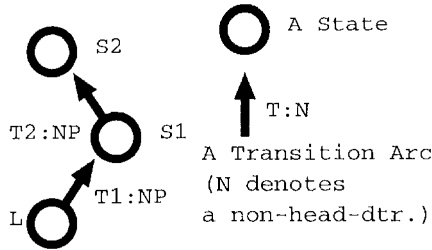


Figure 6: The LA derived from "wrote"

n has a counter part in F_2. More precisely, it produces the unification results for a node n in F_1 such that

- there is a path $p \in paths(n, F_1)$ such that the node reached by p is also defined in F_2, or
- there is a path $p \in paths(n, F_1)$ prefixed by some $p_r \in rs$ or $\langle goals \rangle$.

Note that a node is unified if its structure-shared part has a counter-part in F_2. Intuitively, the routine produces unified results for the part of F_1 *instantiated* by F_2. The other part, that is not produced by *p_unify*, is not required at Phase 2 because it is already computed in a state or DFSs in LAs when the LAs are generated. Then, a sign can be obtained by unifying a sub-structure and the corresponding core-structure.

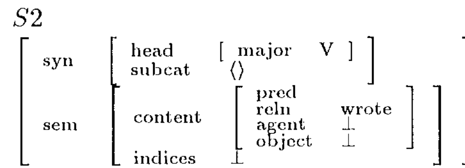
5 Example

This section describes the parsing process of the sentence "My colleague wrote a good paper." The LA generated from the lexical entry for "wrote" in Figure 5 is given in Figure 6. The transition arc $T1$ between the states L and $S1$ is generated by the rule schema in Figure 1. Note that the query to DCP, $freeze([1], append([1], [2], [3]))$, is used to obtain *union* of *indices* values of daughters and the result is *written* to the *indices values* of the mother sign. During the generation of the transition arc, since the first argument of the query is $[\ \bot\ ]$, it is frozen. The core-structures and the *dependency-analyzed* DFSs that augment the LA are shown in Figure 7. We assume that we do not use any restriction, i.e., for any lexical entry l and rule schemata R, $sub(l) = [\bot]$ and $sub(fs(R)) = [\bot]$.

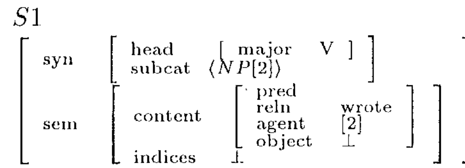
Note that, in the DFSs, the already raised feature structures are eliminated and, that the DFS of the transition arc T contains the *frozen* query as the `goals`.

Assume that the noun phrases "My colleague" and "a good paper" are already recognized by a parser. At phase 1, they are checked if they are unifiable to the condition of transition arcs $T1$ and $T2$, i.e., the NPs which are non-head daughters

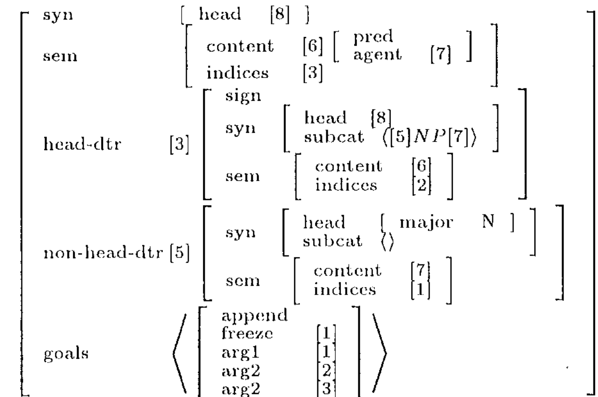
$S2$



$S1$



The dependency-analyzed DFS of $T2$



The dependency-analyzed DFS of $T1$

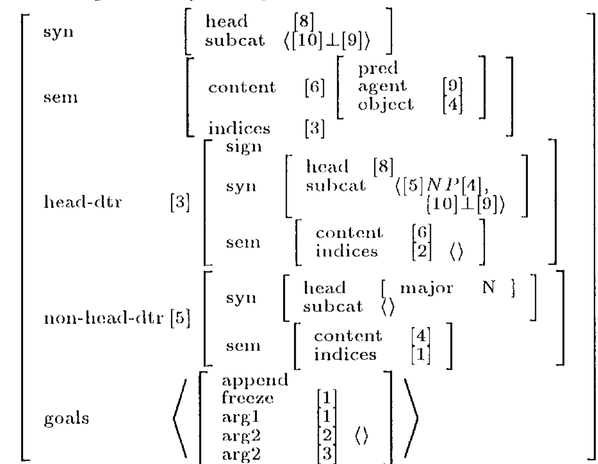
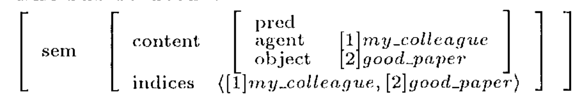
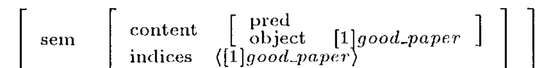


Figure 7: States and DFSs in the LA in Figure 6

The sub-structure for $S2$



The sub-structure for $S1$



The goals,head-dtr,non-head- dtr values are omitted.

Figure 8: The sub-structures obtained in the parsing

Parsing Algorithm	Type of sentences (# of sentences)	Avg Length (word)	Avg Time (sec)
Phase 1 only	all (70)	19.2	1.25 (1.12)
Phase 1 & Phase 2	all (70)	19.2	3.00 (1.65)
Phase 1 & Phase 2	only successful (43)	18.8	3.37 (1.84)
Phase 1 & naive application of rule schemata	only successful (38)	17.13	55.09 (9.27)
Phase 1 & naive application of rule schemata	only successful (5)	31.4	1093.22 (82.12)

A bracketed time indicates non-GC execution time. The experiments was performed on SparcStation 20 with 128 Mb RAM.

Figure 9: Experiments on a Japanese newspaper(Asahi Shinbun)

of $S1$ and $S2$. Since all the unifiability checkings are successful, Phase 1 parsing produces the parse tree whose *form* is presented in Figure 3. The Phase 2 parsing produces the sub-structures in Figure 8. Note that the frozen goals are evaluated and the `indices` values have appropriate values. A parsing result is obtained by unifying the sub-structure for $S2$ with the corresponding core-structure.

The amount of the feature structure nodes generated during parsing are reduced compared to the case of the *naive* application of rule schemata presented in Section 2. The important point is that they contain only either the part in the DFSs that was instantiated by head daughters' sub-structures, and non-head daughters' core-structures and sub-structures, or the part that contributes to the DCP's evaluation. The feature structure that does not appear in a sub-structure appears in the corresponding core-structure. See Figure 7. Because of these properties, the correctness of our parsing method is guaranteed. (Torisawa and Tsujii, 1996).

6 Experiments

We have implemented our parsing method in Common Lisp Object System. Improvement by our method has been measured on 70 randomly selected Japanese sentences from a newspaper (Asahi Shinbun). The used grammar consists of just 5 rule schemata, which are generated from principles and rewriting rules, and 55 default lexical entries given for each part of speech, with 44 manually tailored lexical entries. The total number of states in the LAs compiled from them was 1490. The grammar does not have a semantic part. The results are presented in Figure 9. Our grammar produced possible parse trees for 43 sentences (61.4%). We compared the execution time of our parsing method and a more naive algorithm, which performs Phase 1 parsing with LAs and applys rule schemata to completed parse trees in the naive way described in Section 2. As the naive algorithm caused *thrashing* for storage in GC, it is pointless to compare those figures simply. However, it is obvious that our method is *much* faster than the naive one. We could not measure the execution time for a totally naive algorithm which builds parse trees without LAs because of *thrashing*.

7 Conclusion

We have presented a two-phased parsing method for HPSG. In the first phase, our parser produces parse trees using Lexical Entry Automata compiled from lexical entries. In the second phase, only the feature structures which must be computed dynamically are computed. As a result, amount of the feature structures unified at parsing-time is reduced. We also showed the effect of our optimization techniques by a series of experiments on a real world text.

It can be noticed that each transition arc of the compiled LAs can be seen as a rewriting rule in CFG (or a dotted notation in a chart parser.) We believe this can open the way to integrate several methods developed for CFG, including the inside-outside algorithm for grammar learning or disambiguation, into an HPSG framework. We also believe that, by pursuing this direction for optimizing HPSG parsers, we can reach the point where grammar learning from corpora can be done with concise and linguistically well-defined core grammar.

References

Bob Carpenter. 1992. *The Logic of Typed Feature Structures*. Cambridge University Press.

Keiko Horiguchi, Kentaro Torisawa, and Jun'ichi Tsujii. 1995. Automatic acquisition of content words using an HPSG-based parser. In *NLPRS'95*.

Robert Kasper, Bernd Kiefer, Klaus Netter, and K. Vijay-Shanker. 1995. Compilation of HPSG to TAG. In *ACL 95*.

Carl Pollard and Ivan A. Sag. 1987. *Information-Based Syntax and Semantics Vol.1*. CSLI lecture notes no.13.

Carl Pollard and Ivan A. Sag. 1993. *Head-Driven Phrase Structure Grammar*. University of Chicago Press and CSLI Publications.

Stuart C. Shieber. 1985. Using restriction to extend parsing algorithms for complex feature based formalisms. In *ACL85*.

Kentaro Torisawa and Jun'ichi Tsujii. 1996. Offline raising, dependency analysis and partial unification. In *Third International Conference on HPSG*. In the proceedings of TALN '96.