Factoring Surface Syntactic Structures

Alexis Nasr
LATTICE-CNRS (UMR 8094)
Université Paris 7
alexis.nasr@linguist.jussieu.fr

Mots-clefs – Keywords

Syntaxe de surface, représentation compacte de l'ambiguïté syntaxique, analyse syntaxique.
Surface Syntax, efficient representation of syntactic ambiguity, parsing.

Résumé - Abstract

Nous décrivons dans cet article une façon de factoriser plusieurs structures syntaxiques de surface en un nouvel objet formel : les structures syntaxiques de surfaces factorisées. Ces dernières permettent de ne représenter qu'une seule fois les parties communes à un ensemble de structures syntaxiques de surface. L'article décrit aussi un format nouveau de description des règles d'ordre linéaire de la TST et propose enfin un algorithme efficace pour la réalisation de la transition Morphologie Profonde → Syntaxe de surface.

The aim of this paper is threefold. It describes a way of factoring [1] several Surface Syntactic Structures [= SSyntS] into a new formal object that we call a Packed Surface Syntactic Structure. It also proposes an alternate way of representing the word order rules of the MTT and, eventually, describe an efficient algorithm to perform the Deep Morphological → Surface Syntax transition.

1 Introduction

The reason to introduce Packed Surface Syntactic Structure [= PSSyntS] in the MTT comes primarily from the problem of *intermediate ambiguity* that appears during the Text → Meaning [= $T \rightarrow M$] transition. The most straightforward way to consider the $T \rightarrow M$ transition is to apply to a structure of a given level the rules of the component of the higher level, as well as information from the lexicon, to give birth to one (or several) structures of the higher level. The new structures are then themselves used to build one (or several) structures of the

[1] We shall use here the term *factoring* quite loosely to denote the idea of representing once common parts of a collection of objects.

next higher level. Such a view of the $T \to M$ process is in accordance with the equative nature of the MTT in which the $T \to M$ process is viewed as a series of sub-processes where one sub-process (the realisation of one transition) takes as input the output of the preceding sub-process. This approach suffers a severe drawback from a computational point of view, which is the phenomenon of intermediate ambiguity. We will illustrate it on the DMorphS $\to$ SSyntS transition. Suppose we have the DMorphS $\mathcal{S}$ corresponding to the following string[2] $n_0\ v_1\ n_2\ p_3\ n_4\ p_5\ n_6$, where n, v and p stand respectively for the morpho-syntactic categories *noun*, *verb* and *preposition* while the subscript indicates the position in the string. From a strict surface syntactic point of view, this DMorphS is ambiguous and can correspond to 5 different SSyntS, represented in figure 1. In other words, the application of the SSynt component rules to $\mathcal{S}$ gives rise to the five SSyntS of figure 1. Some of these SSyntS might be spurious and might therefore be discarded later during the $T \to M$ transition. This is the reason why this ambiguity is called intermediate ambiguity: it only appears at intermediate stages of the process. The fact that some SSyntS will be discarded at a later stage is actually an important hypothesis of the work presented here and we will say more about it in section 5.

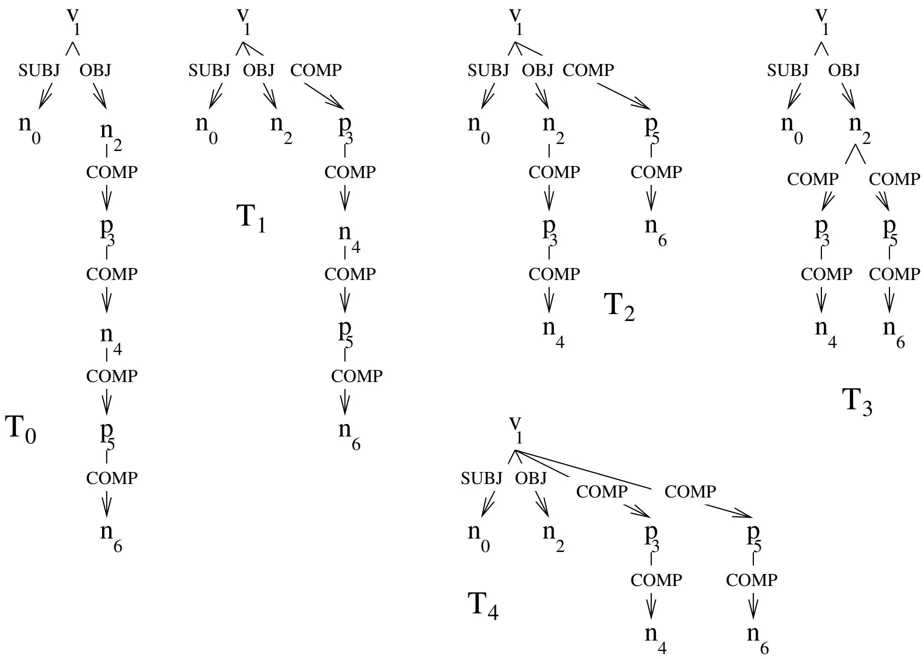


Figure 1: Five Surface Syntactic Structures

The simple solution to this effect of intermediate ambiguity is to continue the $T \to M$ process independently on each SSyntS produced. Such a solution is not very convincing from an economical point of view since, as one can see on figure 1, some structures share important subparts which will be treated independently in each of the SSyntS, the same work will be therefore carried out several times. The problem is actually not only a problem of efficiency, since the intermediate ambiguity grows exponentially(Nasr *et al.*, 2002) with respect to the length of the sentence, in the worst case. Processing independently every SSyntS is therefore not realistic from a computational point of view.

Several solutions can be considered to solve this problem, one of them, which has been explored

[2] We have only kept the morpho-syntactic categories of the words which make up the sentence and got rid of words, like determiners, that are not relevant for the point discussed.

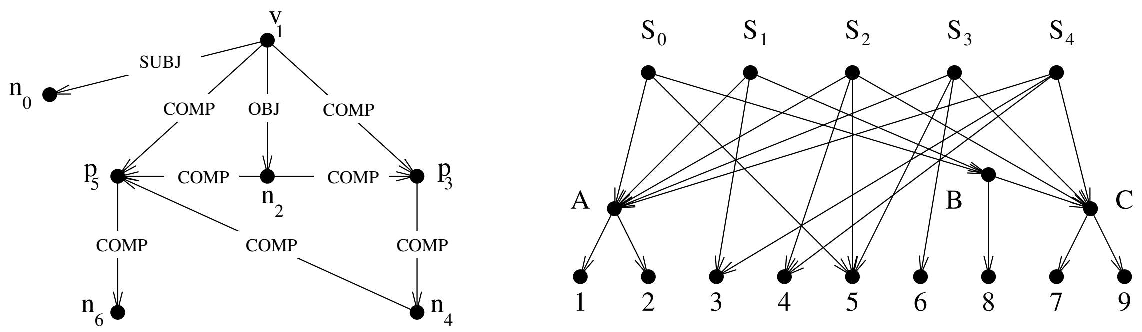
in (Nasr, 1996) was to take into account during the DMorphS $\rightarrow$ SSyntS information that are represented elsewhere in the MTM (like the governent pattern of lexemes and possibly some semantic features of lexemes) in order to give rise only to the "correct" DSyntS. The solution which is explored in this paper is different, it consists in factoring the common parts of the different SSyntS. The result of this factoring is a new kind of structure called a PSSyntS which will be the input of the next transition process, here, the SSynt $\rightarrow$ DSynt transition. The idea of representing, in case of ambiguity, several structures in a single one of a new kind and then process this structure is not new in computational linguistics, it has been used, in particular, in Machine Translation (Dymetman & Tendeau, 2000; C.Emele & Dorna, 1998).

The idea of factoring which was introduced above to solve the problem of intermediate ambiguity can also be justified from the point of view of parsing (realization of the DMorphS $\rightarrow$ SSyntS transition). This fact has been acknowledged a long time ago by researches in the domain of parsing with ambiguous context free grammars. Efficient parsing algorithms for ambiguous grammars as (Younger, 1967) or (Earley, 1970) propose some kind of factoring of the syntactic trees that they build. This factoring is the key to the efficiency of such parsers since it allows the parser to build analyses of sub-parts of a sentence, store them and then combine them to constitute analyses of larger sub-parts of the sentence. When two analyses share a common sub-analysis, this common part is not built twice, it is built once for all and might be used several times during the parsing process. Factoring is therefore not a post-parsing process which takes as input several analyses and factor out their common parts, it lies at the heart of the parsing process which builds a factored representation of the analyses of a sentence.

The structure of the paper is straightforward, in section 2, we describe how several SSyntS can be factored in a single PSSyntS. Section 3 introduces a new formalism for representing word order rules, section 4 describes a parsing algorithm that produces PSSyntS and section 5 concludes the paper.

2 Packed Surface Syntactic Structures

We will illustrate the ideas behind the factoring of several SSyntS on the example introduced above: the five SSyntS corresponding to the sequence $\mathcal{S}$. As one can see on figure 1, the five trees share some sub-parts. SSyntS T_0 and T_1, for example, share the subtree rooted by p_3, all SSyntS share the subject dependency while T_0 and T_5 share the subtree of depth one rooted with v_1. The key to the factoring process is to view each of these SSyntS as a *set* of dependencies. Two SSyntS (sets of dependencies) X and Y sharing some sub-parts will have a non-empty intersection which will be represented once and shared by both X and Y. A PSSyntS corresponding to the five SSyntS of figure 1 have been represented in figure 2. The PSSyntS is made of three elements, a *dependency graph*, which appears on the top left position of figure 2, a function that maps the edges of the dependency graph to the vertices of the third element, which we call the *inclusion graph* and which appears at the top right position of figure 2. The mapping is represented as a table in the bottom part of the figure. The dependency graph corresponds to the union of the five SSyntS. This new structure, which is not a tree anymore, does not keep the information of which subsets of dependencies correspond to an actual SSyntS. This information is represented in the inclusion graph. Each vertex of this graph corresponds to a set of dependencies. The leaves[3] (labelled with integers) correspond to singletons (sets made of a single dependency) while the other vertices correspond to the union of the sets denoted by their daughters. The vertex labelled A, for example, corresponds to the set $\{1, 2\}$, i.e. the set made of the subject and the object dependencies. The vertex S_0 corresponds to the set $\{1, 2, 5, 7, 8, 9\}$ which is exactly the SSyntS T_0 of figure 1. The directed edge which relates vertices S_0 and A indicates that A is a subset of S_0. The function which associates to a vertex of the inclusion graph, a subset of the dependency graph edges will be called the *extension* of the vertex, noted $EXT(\cdot)$. Therefore $EXT(A) = \{1, 2\}$ and $EXT(S_0) = \{1, 2, 5, 7, 8, 9\}$. The vertices of the inclusion graph that correspond to actual SSyntS will be called the *final vertices* of the PSSyntS. Their labels are of the form S_i in order to distinguish them from the non final vertices: A, B and C. The choice of non finite vertices is, for the moment, arbitrary: we could have chosen to define any one of the possible subsets of dependencies that make up the dependency graph. More will be said about such subsets in section 4.



						7	p_3 – COMP → n_4
1	v_1 – SUBJ → n_0	3	v_1 – COMP → p_3	5	n_2 – COMP → p_3	8	n_4 – COMP → p_5
2	v_1 — OBJ → n_2	4	v_1 – COMP → p_5	6	n_2 – COMP → p_5	9	p_5 – COMP → n_6

Figure 2: A Packed Surface Syntactic Structure

More formally, a PSSyntS is a 4-tuple $\langle \mathcal{D}, \delta, \mathcal{I}, \mathcal{F} \rangle$ where $\mathcal{D}$ is the dependency graph, $\mathcal{I}$ is the inclusion graph, δ maps the edges of $\mathcal{D}$ to vertices of $\mathcal{I}$ and $\mathcal{F}$ is the set of final vertices of $\mathcal{I}$. $\mathcal{D}$ itself is a labelled directed graph, defined as a 5-tuple $\langle V_G, \Sigma_S, E_G, \Sigma_L, \mu \rangle$ where V_G is the set of G's vertices, Σ_S is the set of edge labels, which corresponds to surface syntactic labels, E_G is the set of labelled directed edges ($E_G \subseteq V_G \times \Sigma_S \times V_G$), Σ_L is the set of lexical labels and μ is a function that associates a lexical label to the vertices of $\mathcal{D}$. $\mathcal{I}$ is a directed graph defined as a 5-tuple $\langle V_I, \Sigma_I, E_I, \eta, \theta \rangle$ where V_I is the set of vertices, Σ_I is the set of subsets labels, E_I is a set of directed edges ($E_I \subseteq V_I \times V_I$), η maps the elements of V_I on the elements of Σ_I. The function $\theta : V_I \rightarrow \{AND, OR\}$ maps the vertices of $\mathcal{I}$ either to the value AND or to the value OR. We will ignore for the moment this function, its role will become apparent at the end of section 4. $\mathcal{F} \subseteq V_I$ is the set of final vertices. The function EXT is defined recursively:

$$EXT : \quad V_I \rightarrow 2^{E_G}$$

$$EXT(X) = \begin{cases} \{\delta^{-1}(X)\} & \mathit{if}\ \mathit{daughters}(X) = \emptyset \\ \bigcup_{i \in \mathit{daughters}(X)} EXT(i) & \mathit{otherwise} \end{cases}$$

[3]We will use the terminology of trees. When $A \rightarrow B$, we will say that B is a *daughter* of A. We will call a vertex without daughters a *leaf*, and a vertex which is the daughter of no one a *root*.

3 Ordering rules

The Meaning Text Theory distinguishes three kinds of rules (Mel'čuk, 1988) in the Surface Syntactic component. These rules describe the relation between word order, in the Deep Morphological Structure and the surface syntactic dependencies in the corresponding Surface Syntactic Structures. Three kinds of rules are distinguished: *Surface Syntactic rules* or *Syntagm*, which are basically responsible for the ordering of a dependent with respect to its governor, *Pattern for elementary phrases* and *Global word order rules*.

In this work we will only consider word order which is driven by pure surface syntactic consideration, ignoring the influence of the communicative structure of the utterance as well as specific properties of some lexemes, on word order. Furthermore we will only take into account the syntactic constraints that order a word with respect to its governor and its siblings (other dependents of the same governor), which means that we will not be able to take into account non projective structures.

Ordering rules are represented as triplets $\langle C, A, h\rangle$ where C is a morpho-syntactic category, A is an finite state automaton and h is the lexical head of the rule. The automaton describes all the dependents a word h of category C can have. The triplet $\langle V, (\mathrm{SUBJ}, N)\ (\mathrm{HEAD}, v_1)(\mathrm{OBJ}, N)?\ (\mathrm{COMP}, P)*, v_1\rangle$ is an example of such a rule (here the automaton has been represented as an equivalent regular expression). Such a rule says that verb v_1 must have as a dependent a subject[4], of category noun, an optional object also of category noun (the optionality is represented by the $?$ sign) and any number of prepositional complements (the multiplicity is represented by the Kleene star). The pair (HEAD, v_1) does not describe a dependency, it only indicates the position of the head with respect to its dependents[5]. The order between the dependents appearing on the same side of the head corresponds to the linear order of the (function, category) couples in the regular expression. The object, for example, must appear to the left of the prepositional complements.

Ordering rules indicate the possible dependents of a word of a given category. They further indicate which dependents are mandatory, which are optional and which can be repeated. They eventually define a linear order between them. This way of representing the relation between word order and the surface syntactic structure of an utterance is not meant to replace the different rules of the surface syntactic component which have been mentionned above. They should rather be seen as the result of an automatic transformation of these rules in a format which is more suitable to an automatization of the DMorph $\rightarrow$ SSynt transition.

The automata corresponding to the verb (A_{v_1}), to the noun (A_{n_2}) and to the preposition (A_{p_5}) rules have been represented graphically in figure 3. Each transition of the automata is labelled by a couple $\langle F, C\rangle$ where F is a Surface Syntactic Function label (or the special label HEAD) and C is a morpho-syntactic category. These automata are acceptors/generators of languages, in accordance with standard automata theory (Hopcroft & Ullman, 1979). The languages are, as usual, sets of words and each word is a string of couples $\langle F, C\rangle$. Each of these words will be interpreted in our framework as a Surface Syntactic tree of depth one, as shown on the bottom of figure 3. Each couple $\langle F, C\rangle$ represents a dependency, apart from the couple $\langle \mathrm{HEAD}, X\rangle$, which indicates the lexical item that labels the root. Such automata can therefore be seen as *tree generating* automata.

[4]Surface syntactic funtions are inspired by (Mel'čuk & Pertsov, 1987).

[5]One can notice that the lexical head of a rule is represented twice, once as the third element of the triplet and once as a transition of the automaton. This redundancy was introduced for notational convenience.

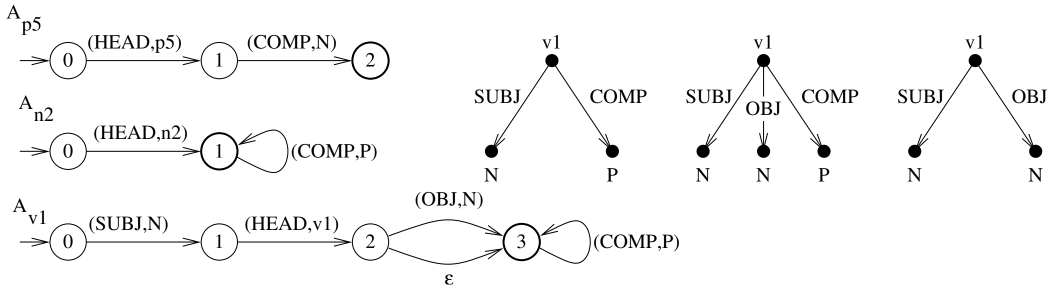


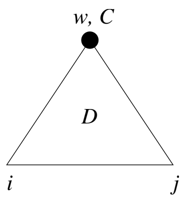
Figure 3: Three ordering rules and some generated trees

4 Parsing

The parsing algorithm is a simple adaptation of the well known CYK algorithm (Younger, 1967). The key idea of this algorithm is to analyse the sub-part $[w_i, w_j]$ of a sentence by combining the analyses of sub-part $[w_i, w_k]$ with the analysis of sub-part $[w_{k+1}, w_j]$ (with $i \leq k < j$). The efficiency of the algorithm comes from the fact that the sub-analyses are computed once for all and stored in a two dimensional table. The order in which sub-parts of a sentence are analysed is therefore important: before analysing sub-part $[w_i, w_j]$ the two sub-parts $[w_i, w_k]$ and $[w_{k+1}, w_j]$ must have been analysed for all possible values of k.

Two parsing algorithms will be described, in section 4.1 and 4.2. Both build PSSyntS but the first one is exponential in the worst case, while the second is polynomial. The two algorithms have a lot in common, this is why this section will split into two sub-sections near the end. Everything that will be said before is true of both parsers.

The input of the parser is a couple $\langle W, R \rangle$ where $W = [w_1, w_n]$ is the sequence of words that make up the sentence to be parsed and $R = [r_1, r_n]$ is a sequence of rules such that r_i is the rule corresponding to w_i, or, in other words, w_i is the head of r_i. The output of the parser is a PSSyntS $\langle \mathcal{D}, \delta, \mathcal{I}, \mathcal{F} \rangle$. The SSyntS of sentence W are the extensions of the elements of $\mathcal{F}$.



Central to the parsing algorithm is the notion of *sub-analysis*. A sub-analysis is a SSyntS which corresponds to a given sub-part of the sentence and which is rooted by a given word. A sub-analysis made of the dependency set D, rooted with word w of category C and corresponding to the sub-part $[w_i, w_j]$ of the sentence is depicted on the left.

More formally, a sub-analysis is a 5-tuple $\langle\langle C, A, w_l \rangle, s, i, j, S\rangle$ where $\langle C, A, w_l\rangle$ is the rule r_l (the rule that has w_l as head), s is a state of automaton A. Such a state is called the *current state* of the sub-analysis. i, j are two integers indicating that the sub-analysis corresponds to the part of the sentence $[w_i, w_j]$, S is a vertex of the inclusion graph of the PSSyntS. This vertex denotes the set of dependencies that make up the sub-analysis, in other words, the SSyntS which corresponds to the sub-analysis is $EXT(S)$. The root the SSyntS is the word w_l. If s is an accepting state of A, we will say that the sub-analysis is *saturated*, which means that all its mandatory dependents have been identified.

The algorithm is based on three operations that construct sub-analyses :

- The operation of *merging* combines the two sub-analysis $X_1 = \langle\langle C_1, A_1, w_l\rangle, s_1, i, k, S_1\rangle$ and $X_2 = \langle\langle C_2, A_2, w_m\rangle, s_2, k{+}1, j, S_2\rangle$ to form the new sub-analysis $X_3 = \langle\langle C_1, A_1, w_l\rangle,$

$s_3, i, j, S_3\rangle$. This operation can be paraphrased in the following way: X_1 corresponds to a sub-analysis of the sequence $[w_i, w_k]$ and X_2 corresponds to a sub-analysis of $[w_{k+1}, w_j]$, X_2 is saturated or, in other words, s_2 is an accepting state of A_2 and X_1 is waiting for a dependent of category C_2 (there is a transition in A_1 from state s_1 to state s_3 labelled with category C_2), which is also the category of the root of S_2. During this operation, the dependency graph $\mathcal{D}$ and the inclusion graph $\mathcal{I}$ are updated in the following way :

- A new edge $w_l \to w_m$ is added (if not already present) to $\mathcal{D}$. In the case the dependency $w_l \to w_m$ did not exist in $\mathcal{D}$, a new vertex x is added to $\mathcal{I}$ and the value of $\delta(w_l \to w_m)$ is set to x.
- A new vertex S_3 is added to $\mathcal{I}$ as well as the three edges $S_3 \to S_1$, $S_3 \to S_1$ and $S_3 \to x$ which means that $EXT(S_3) = EXT(S_1) \cup EXT(S_2) \cup EXT(x)$. If S_3 corresponds to a sub-analysis of the entire sentence ($i = 1$ and $j = n$) then S_3 is added to the set $\mathcal{F}$ of the final vertices of $\mathcal{I}$.

- The operation of *leftmost attachment* combines a sub-analysis $X_1 = \langle\langle C_1, A_1, w_l\rangle, s_1, i, j, S_1\rangle$ and a rule $\langle C_2, A_2, w_m\rangle$ to create a sub-analysis $X_2 = \langle\langle C_2, A_2, w_m\rangle, s_2, i, j, S_2\rangle$. X_1 must be saturated (s_1 is an accepting state of A_1) and A_2 is such that there is a transition from its initial state to state s_2 labelled with category C_1. During this operation, the dependency graph $\mathcal{D}$ and the inclusion graph $\mathcal{I}$ are modified in the following way :

  - As in the case of merging, a new edge $w_l \to w_m$ is added (if not already present) to $\mathcal{D}$ and a new vertex x is potentially added to $\mathcal{I}$.
  - A new vertex S_2 is added to $\mathcal{I}$ as well as the two edges $S_2 \to S_1$ and $S_2 \to x$ ($EXT(S_2) = EXT(S_1) \cup EXT(x)$).

  This operation corresponds to the creation of the leftmost dependency of word w_m.

- The operation of *head recognition* combines a sub-analysis $X = \langle\langle C, A, w_l\rangle, s, i, j, S\rangle$ and the word w_l. This operation corresponds to the recognition of the head of rule $\langle C, A, w_l\rangle$. This operation happens when X's current state (s) is such that there is a transition labelled (HEAD, w_l) going out of it. This operation does not modify the dependency graph, nor the inclusion graph.

4.1 The exponential parser

The parsing algorithm consists in iteratively building sub-analyses corresponding to larger and larger sub-parts of the sentence using the three operations described above. For the sake of clarity, we will note $S_{i,j}$ a sub-analysis corresponding to the sub-part $[w_i, w_j]$ of the sentence. Each time a sub-anlysis $S_{i,j}$ is built, it is stored in the cell $\mathcal{T}_{i,j}$ of a two dimensional table $\mathcal{T}$. $\mathcal{T}$ is filled in such a way that $\mathcal{T}_{i,j}$ is filled after both $\mathcal{T}_{i,k}$ and $\mathcal{T}_{k+1,j}$ (with $i \leq k < j$) have been filled. Filling cell $\mathcal{T}_{i,j}$ amounts to combining the sub-analyses of $\mathcal{T}_{i,k}$ and $\mathcal{T}_{k+1,j}$ with the merging operation and then possibly adding new sub-analyses to $\mathcal{T}_{i,j}$ with the leftmost attachment and the head recognition operation. When all the cells of $\mathcal{T}$ have been filled, if the set $\mathcal{F}$ of final nodes of the PSSyntS is not empty, the extension of each of its elements corresponds to a SSyntS of the sentence. The process is initiated by adding in every cell $\mathcal{T}_{i,i}$ the word w_i which will cause the first head detection operations.

The result of parsing sequence $s_1 = [n_0\ v_1\ n_2\ p_3\ n_4\ p_5\ n_6]$ using rules $[\langle N, A_{n_0}, n_0\rangle, \langle V, A_{v_1}, v_1\rangle, \langle N, A_{n_2}, n_2\rangle, \langle P, A_{p_3}, p_3\rangle, \langle N, A_{n_4}, n_4\rangle, \langle P, A_{p_5}, p_5\rangle, \langle N, A_{n_6}, n_6\rangle]$ is shown in the table of figure 4. Each entry of cell (i, j) of the table shows one operation that lead to a sub-analysis of $[w_i, w_j]$. The type of operation (M for merging, H for head recognition and L for leftmost attachment) is represented between square brackets and the effect of the operation on the inclusion graph is represented as an equation of the form $W = X \cup Y \cup Z$ which is to be interpreted as: vertex W have been added to $\mathcal{I}$ as well as the three edges $W \to X$, $W \to Y$ and $W \to Z$.

0	1	2	3	4	5	6	
$A = \emptyset \cup 1 [L]$ $\emptyset$ $[H]$ n_0	$A[H]$	$P = A \cup 2 \cup \emptyset [M]$		$G = A \cup 2 \cup D [M]$ $H = P \cup 3 \cup B [M]$		$S_0 = A \cup 2 \cup I\ [M]$ $S_3 = A \cup 2 \cup J\ [M]$ $S_1 = P \cup 3 \cup F [M]$ $S_2 = G \cup 4 \cup C [M]$ $S_4 = H \cup 4 \cup C [M]$	0
	v_1						1
		$\emptyset$ $[H]$ n_2		$D = \emptyset \cup 5 \cup B\ [M]$		$I = \emptyset \cup 5 \cup F\ \ [M]$ $J = D \cup 6 \cup C\ [M]$	2
			$\emptyset[H]$ p_3	$B = \emptyset \cup 7 \cup \emptyset\ \ [M]$		$F = \emptyset \cup 7 \cup E\ \ [M]$	3
				$\emptyset$ $[H]$ n_4		$E = \emptyset \cup 8 \cup C\ \ [M]$	4
					$\emptyset[H]$ p_5	$C = \emptyset \cup 9 \cup \emptyset\ \ [M]$	5
						$\emptyset$ $[H]$ n_6	6

Figure 4: The parsing table at the end of the process

4.2 The polynomial parser

The parsing process described above builds a PSSyntS which corresponds to a set of SSyntS in such a way that every dependency is represented exactly once in the dependency graph. But one can notice that the cell $\mathcal{T}_{0,6}$ of the table contains five entries, each one corresponding to one of the five SSyntS of figure 1. This is due to the fact that, by definition of the algorithm, when parsing a sentence $\mathcal{S}$ of length n, there will be as many entries in cell $\mathcal{T}_{0,n-1}$ as there are SSyntS corresponding to sentence $\mathcal{S}$. This situation is not satisfactory since, as we already said, the number of SSyntS is, in the worst case, an exponential function of n, so will be the number of entries in $\mathcal{T}_{0,n-1}$, the algorithm will therefore be exponential. Fortunately, we can avoid such an exponential growth by restricting the number of sub-analyses in each cell of the table. The restriction stems from the following remark: when two sub-analyses share the same rule and the same current state, they are interchangeable: one can be replaced by the other in any larger sub-analysis. We can therefore represent these two sub-analyses as a single object which denotes the two sub-analyses. Let's illustrate this on an example. The two sub-analyses of $\mathcal{T}_{2,6}$ are rooted with word n_2 and both are saturated (their current state is an accepting state of automaton A_N). They correspond to the two subtrees of T_0 and T_3 (in figure 1) which have n_2 as a root. These two sub-analyses differ only in the node they correspond to in the inclusion graph: I in one case and J in the other. Instead of representing both sub-analyses as separate objects, we will represent them as one object which corresponds to two vertices of the inclusion graph. Instead of having the two sub-analyses $\langle\langle N, A_N, n_2\rangle, 1, 2, 6, I\rangle$ and $\langle\langle N, A_N, n_2\rangle, 1, 2, 6, J\rangle$, we will have the single sub-analysis $\langle\langle N, A_N, n_2\rangle, 1, 2, 6, \{I, J\}\rangle$. As a consequence, the two entries of

$\mathcal{T}_{0,6}$, S_0 and S_3, corresponding to SSynsS T_0 and T_3 of figure 1 will now be represented as a single entry.

A sub-analysis can now denote more than one SSyntS or, to say in another way, can correspond to more than one vertex in the inclusion graph. We will therefore have to manipulate sets of vertices and introduce some formal means to represent them in the inclusion graph. This means is function θ, introduced as a component of the inclusion graph in section 2, and unused since. This function maps the vertices of the inclusion graph either to the value AND or to the value OR. In the first case, we will call the vertex an AND-vertex and, in the second, an OR-vertex. The difference between these two types of vertices lies in the semantics of the relation which exists between a vertex and its daughters. An AND-vertex denotes a set which is the union of the sets denoted by its daughters. It is this semantics that we have used until now and which was materialized by the EXT function ($EXT(X) = \bigcup_{i \in daughters(X)} EXT(i)$). An OR-vertex denotes a set of sets which is the set composed with the sets denoted by the daughters of the vertex. The binary operation which makes a set out of two elements is represented by symbol $\sqcup$ ($a \sqcup b = \{a, b\}$). The EXT function is now defined as follows:

$$EXT: \quad V_I \rightarrow 2^{2^{E_G}}$$

$$EXT(X) = \begin{cases} \{\delta^{-1}(X)\} & \textit{if } daughters(X) = \emptyset \\ \bigcup_{i \in daughters(X)} EXT(i) & \textit{if } \theta(X) = AND \\ \bigsqcup_{i \in daughters(X)} EXT(i) & \textit{if } \theta(X) = OR \end{cases}$$

The modifications that need to be done in the parsing algorithm to take into account this change in the inclusion graph are minimal. Whenever a sub-analysis $S = \langle R, s, i, j, I \rangle$ is to be added to the cell $\mathcal{T}_{i,j}$ of the parsing table, before adding it, a lookup is done through the entries of the cell. If a sub-analysis $S' = \langle R, s, i, j, I' \rangle$ having the same rule as S and the same current state is found, then S is not added to the cell. Two situations can occur, either I' is an OR-vertex of the inclusion graph or I' is an AND-vertex. In the first case, I is added as a daughter of I'. In the second case, a new OR-vertex I'' is added to the inclusion graph, I and I' are added as I'' daughters and S' is changed to $\langle R, s, i, j, I'' \rangle$ (I' is replaced by I''). The rest of the algorithm is unchanged.

5 Conclusion

The main point of this paper was to introduce Packed Surface Syntactic Structures, a new formal object in the MTT. A PSSyntS represents the different SSyntS of a sentence. The key idea behind PSSyntS is to represent only once dependencies that are shared in the SSyntS. This factoring allows the definition of a parsing algorithm that performs in polynomial time even if the number of parses is exponential. This result couldn't have been obtained without a efficient representation of ambiguity. But for this advantage not to be lost, the PSSyntS which has been produced should not be unpacked (opening Pandora's box), the other transitions of the MTT, (SSyntS $\rightarrow$ DSyntS and DSyntS $\rightarrow$ SemS) should therefore be performed on packed structure, using, for example, formal tools introduced in (Rozenberg, 1997) and already used in the framework of the MTT in (Bohnet & Wanner, 2001). The hypothesis which is made here is that some of the SSyntS represented in the PSSyntS produced the parser are ill-formed and that this ill formedness will be revealed at later stages of the $T \rightarrow M$ and at the end of the process, few SemS will have survived. This hypothesis is, as its name suggests, only a hypothesis. We do not know at the moment to which extent the ambiguity will decrease at later stages of the $T \to M$ process. Some obvious examples come immediately to mind, as the government patterns of lexemes which, when applied to the PSSyntS will rule out certain SSyntS[6]. In the sentence $\mathcal{S}$, the actual lexeme v_1 for example might not licence prepositional complements, which will rule out structures T_1, T_2 and T_4. But it is unclear for the moment where, in an MTM, are the other constraints that will rule out other (syntactically, lexically, semantically, communicatively ...) ill-formed analyses and how they can be used in the $T \to M$ transition.

References

BOHNET B. & WANNER L. (2001). On using a parallel graph rewriting grammar formalism in generation. In *8th European Natural Language Generation Workshop at the Annual Meeting of the Association for Computational Linguistic*, Toulouse.

C.EMELE M. & DORNA M. (1998). Ambiguity preserving machine translation using packed representation. In *COLING-ACL 1998*, Montreal.

DYMETMAN M. & TENDEAU F. (2000). Context-free grammar rewriting and the transfer of packed linguistic representation. In *Proceedings of the 19th International Conference on Computational Linguistics (COLING'00)*, p. 1016–1020, Saarbrücken.

EARLEY J. (1970). An efficient context-free parsing algorithm. *Communcations of the ACM*, **8**(6), 451–455.

HOPCROFT J. & ULLMAN J. (1979). *Introduction to Automata Theory, Languages and Computation*. Reading, Massachussetts: Addison-Wesley.

MEL'ČUK I. A. (1988). *Dependency Syntax: Theory and Practice*. New York: State University of New York Press.

MEL'ČUK I. A. & PERTSOV N. V. (1987). *Surface Syntax of English*. Amsterdam/Philadelphia: John Benjamins.

NASR A. (1996). *Un modèle de reformulation automatique fondé sur la Théorie Sens Texte: Application aux langues contrôlées*. PhD thesis, Université Paris 7.

NASR A., RAMBOW O., CHEN J. & BANGALORE S. (2002). Context-free parsing of a tree adjoining grammar using finite-state machines. In *Proceedings the Sixth Workshop on Tree Adjoining Grammars (TAG+ 6)*, Venise, Italie.

G. ROZENBERG, Ed. (1997). *Handbook of Graph Grammars and Computing by Graph Transformation Volume 1: Foundations*. World Scientific.

YOUNGER D. (1967). Recognition and parsing of context-free grammar in time n^3. *Information and Control*, (10), 189–208.

[6]One could argue that this piece of information should be taken into account in the ordering rules.