A Generative Approach to Parsing in the Framework of the Meaning-Text Theory

Alexis Nasr*

LATTICE-TALANA
Université Paris 7
and
Observatoire de Linguistique Sens-Texte
alexis.nasr@linguist.jussieu.fr

1 Introduction

Our study is concerned with the procedural aspects of the Text $\Rightarrow$ Meaning correspondence (T $\Rightarrow$ M) in the Meaning-Text Theory (MTT), or, strictly speaking, the construction of the Semantic Representation of a sentence starting with its Surface Phonetic Representation. More specifically we will be dealing with the realization of the transition between a Deep Morphological Structure (DMorphS) and its corresponding Surface Syntactic Structures (SSyntS)[1]. This transition corresponds to the common task of syntactic analysis, or *parsing*; for the sake of simplicity, it will be referred to, in this paper, as parsing.

The original aspect of this work with respect to the MTT is our choice to devise a generative procedure in the framework of the MTT. This choice might seem surprising since the MTT is a non generative theory. It has therefore appeared important to us to explain the reasons of this choice. This will be done in Section 2. The structure of the remaining of the paper is straightforward, Section 3 describes the formalism: its elementary objects, a combining operation and a generative procedure. The parsing algorithm is described in Section 4.

*The author would like to thank all the participants to the symposium which took place in Saint Just during the summer of 1999 for their fruitful comments on earlier versions of this paper.

[1]We will ignore in this study the communicative, anaphoric and prosodic structures.

2 From an equative to a generative approach to parsing

The task of parsing, and more generally, the realization of the T $\Rightarrow$ M transition in the framework of the MTT has not been much studied and a number of important choices need to be made when considering this task. We will discuss in this section what we consider to be the most important of these choices: to choose between an equative and a generative approach to parsing. We will first describe what we believe to be the most natural way of considering parsing in the MTT : the equative approach. We will show that this approach suffers some important drawbacks from a procedural point of view and propose one way to overcome them.

In the equative approach, the T $\Rightarrow$ M transition can be seen as a series of transitions of a structure of a given representation level to a structure of the deeper level. If we take a closer look at the transition we are interested in here, the DMS $\Rightarrow$ SSyntS transition, we observe that such a transition makes use of several pieces of informations which are scattered in several places in a Meaning Text Model (MTM). More precisely, this transition needs rules contained in the Surface-Syntactic Component: surface syntactic rules (or syntagms), patterns for elementary phrases and global word order but it also needs informations contained in the Explanatory Combinatorial Dictionary (ECD)[MP87b]: government patterns of lexemes and lexical functions. More generally, it is supposed that all rules and representations can and must be consulted during all transitions.

Devising a way to carry out the DMS $\Rightarrow$ SSyntS transition with such an organization of the linguistic knowledge naturally leads to a complex procedure. Without entering into the details of such a procedure, let us just mention that it must take into account "rules" of very different natures which are represented in different formalisms and that the rules should be applied synchronously.

Our main goal being to propose an efficient automatic realization of the DMS $\Rightarrow$ SSyntS transition, we will allow ourselves to modify the representation of the linguistic knowledge of the MTT when its organization does not favor a simple procedural schema. In this work, these modifications take the form of a *grouping* of the different knowledge sources necessary to perform the DMS $\Rightarrow$ SSyntS transition. This grouping leads to the definition of a complex object called *elementary tree*, described in Subsection 3.1.

It is important at this point to note that our perspective on the MTT is quite different from the perspective that has been defined in [Mel88] which explicitly decided to ignore the dynamic (or procedural) aspects of the Meaning $\Leftrightarrow$ Text transitions, considering that they lie outside the scope of linguistics. These procedural aspects lie, on the contrary, at the heart of our approach. We will refer to our approach as the *procedural* approach and to [Mel88] approach as the *descriptive* one.

The difference between these two perspectives is well illustrated by the difference of the representation of linguistic knowledge they propose. In the descriptive perspective, different linguistic phenomena are described by different rules, in a analytical approach. As an example, there is no reason, from a descriptive point of view, to merge two different phenomena as agreement and government patterns in a single object, these two phenomena are therefore described separately: agreement is described in surface syntactic rules while the government pattern of a lexeme is described in its entry, in the ECD. In the procedural approach, these phenomena are not described separately; they are merged into a single homogeneous object in order to be taken into account simultaneously in a single operation during processing. The procedural approach can be described as a synthetic approach.

These two approaches and the different organization of the linguistic knowledge they propose should not be put in a competition, they serve different purposes, it is therefore natural that they differ. The best way to see elementary trees is to consider them as the result of a compiling of an MTM, in the original form. The compiling procedure is unfortunately not defined yet and we do not know at this point if it can be automated.

The grouping process which lead to the definition of elementary trees also had an unexpected consequence: the new structures can be interpreted as the elementary objects of a generative formalism, as will be shown in 3. The compiling process therefore transformed an equative system into a “generative” one. This transformation introduces some concepts which did not exist in the MTT, as the concept of a *language generated by a MTM.* It also opens the way to a new definition of parsing in the MTT, in this new framework. Parsing a sentence S given an MTM is not seen anymore as performing the transition between the Deep Morphological Structure of S and its Surface Syntactic Structures but to decide whether S belongs to the language generated by the MTM by producing one or several ways of deriving S using the rules of the MTM in its generative format. An MTM in this format can therefore be seen as a generative dependency grammar, as the systems proposed in [Hay64, Rob70] and [Gai65].

The approach to parsing we have chosen to explore in this study, the generative one, is not the only possible one; reductionistic approaches, for example, which are independent of any generative procedure have been proposed in [Kar95]. The advantage of the generative approach lies in the enormous amount of work and results about parsing in this framework. Parsing is a complex task and if a shift to the generative framework allows us to use some of the techniques and results already developed, such a shift is worthwhile. This shift has another interesting feature: it can be seen as an attempt to understand the relations between an equative and a generative system.

3 A generative formalism

The generative formalism presented in this section is a tree based, lexicalized one. The elementary structures of the formalism, described in 3.1 are lexico-syntactic trees, as in tree grammars ([GM75]) and in Tree Adjoining Grammars([JLT75], [Jos87]). Each of these elementary structures is associated to a lexical item. Elementary trees are combined together through a unique operation called attachment, described in 3.2. The generative procedure is described in section 3.3.

3.1 Elementary trees

Elementary trees can be seen as under-specified fragments of SSyntS. They are under-specified in the sense that some nodes of the tree are not labeled with a lexical item. An elementary tree describes a possible lexico-syntactic context of a given lexeme, which we will call the *lexical anchor* of the elementary tree, borrowing this term from Lexicalized Tree Adjoining Grammars. The context represented by the elementary tree describes part of the passive and the active valency of the anchor, as well as morphological and lexical dependencies[2] existing between the anchor and other nodes of the elementary tree. Three simple elementary trees corresponding respectively to an adjective, a verb and a preposition are represented in Figure 1. Anchors are represented as black nodes, the label of the dependencies are borrowed from [IM99] and [MP87a].

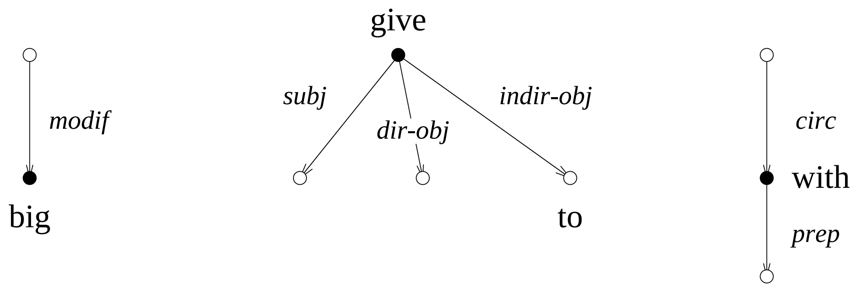


Figure 1: Three simple elementary trees

As shown in Figure 1, depending on the nature of their anchor, elementary trees can exhibit different shapes. A first important dimension along which elementary trees can vary is the representation, in the elementary tree, of the passive valency of the anchor, which is represented only in elementary trees associated

[2]We will use informally the term *lexical dependency* to represent things like governed prepositions, selection of an auxiliary by a past participle as well as syntagmatic lexical function (following the classification of lexical functions introduced in [ART93]).

to lexemes in a modifier position, as the adjective and the preposition in Figure 1. This feature allows to restrict the dependents of the anchor represented in the elementary tree to the active valency of the anchor (the actants of the anchor). The elementary tree associated to a noun, for example, will not contain the dependencies that could link this noun, in a SSyntS, to its modifiers. These dependencies are represented in the elementary trees of the modifiers.

The second fundamental difference concerns the size of the elementary tree (its *domain of locality* in Tree Adjoining Grammars terminology) which extends to all the nodes of the SSyntS entering in a morphological, lexical or semantic dependency with the anchor. Elementary trees can therefore exhibit extended domains of locality as shown in the French example of Figure 2. In this example, the elementary tree associated to the past participle of the verb *donner* contains five other nodes: the auxiliary, since the past participle constrains its lexical value, the governed preposition *à* as well as the subject, the direct and the indirect objects because of the semantic dependency existing between each of them and the past participle.

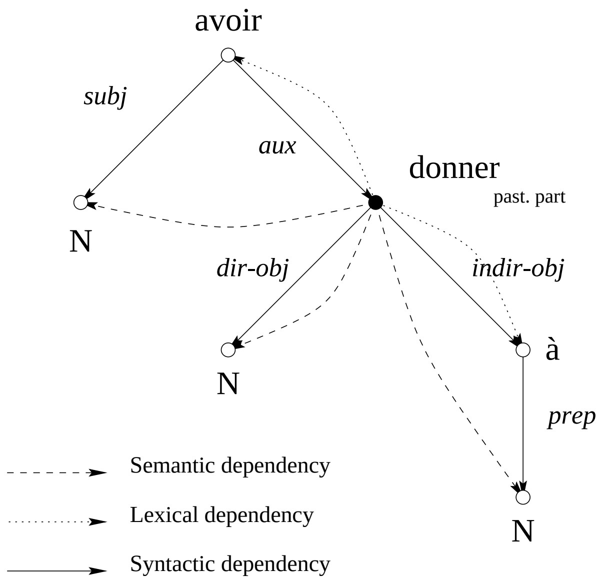


Figure 2: An extended elementary tree

Elementary trees are represented as feature structures in which syntactic dependencies are represented as complex features. The elementary tree of Figure 2 has been represented as a feature matrix in Figure 3. The features names are self explanatory, the feature *anchor* indicates if a node is the anchor of an elementary tree. We will not get into the details of this representational tool (see for example [Shi86]) although we will use some non conventional features - for the representation of repeatable dependencies, for example. A detailed description of these features and the way they are handled during unification, when elementary trees are combined together is an important point but it lies outside the topic of this paper.

$$\begin{bmatrix} \text{lex} = \text{avoir} & \\ \text{cat} = \text{Aux} & \\ \text{subj} = & \begin{bmatrix} \text{cat} = \text{N} \end{bmatrix} \\ \text{aux} = & \begin{bmatrix} \text{lex} = \text{donner} & \\ \text{cat} = \text{V} & \\ \text{mode} = \text{past-part} & \\ \text{anchor} = \text{y} & \\ \text{dir-obj} = & \begin{bmatrix} \text{cat} = \text{N} \end{bmatrix} \\ \text{indir-obj} = & \begin{bmatrix} \text{lex} = \text{à} & \\ \text{cat} = \text{Prep} & \\ \text{prep} = & \begin{bmatrix} \text{cat} = \text{N} \end{bmatrix} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Figure 3: An elementary tree as a feature matrix

3.1.1 Word order

The most challenging linguistic information to represent in elementary trees and maybe in any dependency generative grammar is the description of grammatical word order. Contrary to syntagmatic trees, in which word order and syntactic structure are closely linked, the dependency framework allows the linguist to represent the syntactic structure of an utterance independently of the actual linear order of its words. At this point, it is important to sharpen our terminology. The term *dependency tree* will designate an unordered dependency tree; ordered trees will be explicitly referred to as *ordered dependency trees.* A total ordering of the nodes of a given tree will be referred to as a *linear ordering* of the tree. Hence we have: dependency tree + linear ordering = ordered dependency tree.

There is no direct relation between a dependency syntactic representation of a sentence and the sequence of its words. As a consequence, a given dependency tree can correspond to any linear ordering of its nodes. This feature has two interesting consequences for linguistic description. First, any linear ordering can potentially be associated to a dependency tree, allowing to represent some complex non projective [3] structures. The second advantage is the ability to associate a unique dependency tree to several linear orderings in the case of free word order languages.

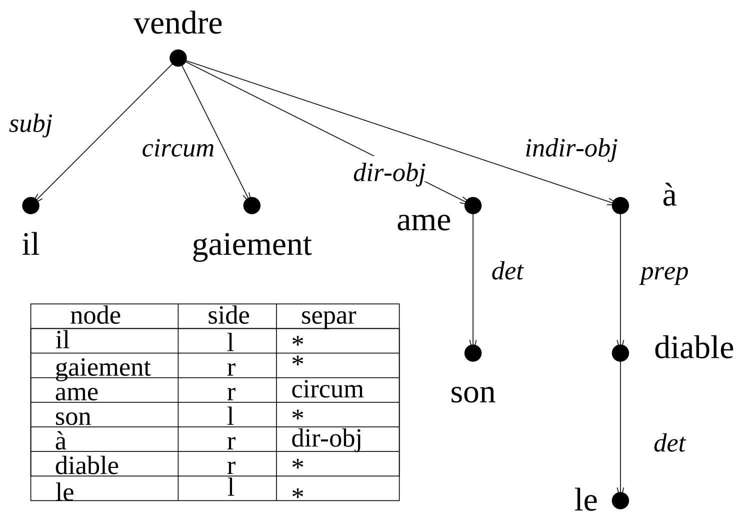
Some dependency formalisms, as [Hay64], merge the representation of syntactic dependencies and linear ordering in a single ordered dependency tree. Such systems usually enforce the projectivity principle which constrains the set of dependency trees that can be generated and avoids the complications attached to the generation of non projective structures (on this topic, see [KNR98]). This solution appears to us as an impoverishment of the dependency formalism since the two advantages mentioned above are lost.

In order to preserve some independence of the syntactic representation and the linear ordering, we will propose a way to encode linear order in dependency tree and to express linear order constraints in the elementary trees based on the notion of *positional features*. This representational means was originally introduced by P. Hellwig ([Hel85, Hel86]) to represent ordered projective dependency trees. We will propose an enriched version of positional features which allows for the representation of partially ordered dependency trees. We will not discuss in this paper the case of non projective structures. The interested reader is referred to [Nas96] for a way to represent some non projective structures by means of positional features.

Positional features give the linear position of a dependency tree node among the other nodes of the tree. Under the constraint of projectivity, this amounts to specifying the position of a node d with respect to its governor g (indicating if d is situated to the left or to the right of g) and its position among the other dependents of g situated on the same side of g. This second piece of information can be represented in several ways; in Hellwig's original format it was represented by a numbering of the siblings: every daughter of a node is associated to a integer which specifies its position among its siblings. We have chosen to represent it by indicating for d its closest sibling s separating d and g, s being referenced by its functional label (the surface syntactic relation labeling the dependency linking s and g). The first feature will be called the `side` feature and the second, the `separ` feature. The closest right or left dependent of a node will have the symbol `*` in its `separ` feature, indicating that it is not separated from its governor by any other dependent of the governor. Thanks to the projectivity principle, these two features allow us to define a single ordering of the nodes. An example of a dependency tree, ordered by means of positional features is

[3] Projectivity is a property defined on the set of ordered dependency trees, originally introduced by [Lec60]. A dependency is *projective* if its dependent is not separated, in the linear sequence, from the governor by anything except descendents of the governor. An ordered dependency tree is said to be projective if all its dependencies are projective.

represented in Figure 4. This tree corresponds to the French sentence *Il vendit gaiement son âme au diable* (literally *He sold cheerfully his soul to the devil*). It is important to note that without the projectivity assumption these two features are not sufficient to order a dependency tree. For the sake of readability, the positional features have been displayed in a table.

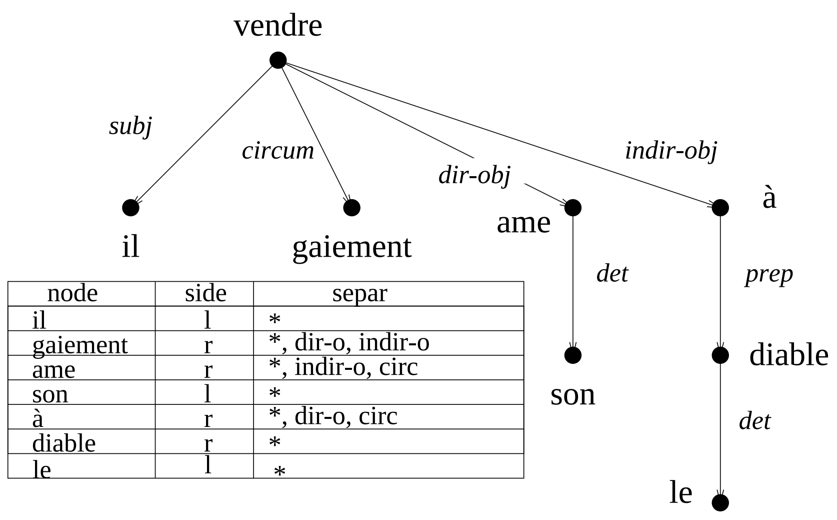


node	side	separ
il	l	*
gaiement	r	*
ame	r	circum
son	l	*
à	r	dir-obj
diable	r	*
le	l	*

Figure 4: An ordered dependency tree

In order to represent partially ordered dependency trees, we will under-specify the two positional features. When the position of a node d with respect to its governor g is unconstrained, i.e. d can either appear to the left or to the right of g, the `side` feature of d will not be specified. The `separ` feature will give the list of the *possible* closest separating siblings of d among its siblings. An example of a partially ordered tree is given in Figure 5. In this example, the three right dependents of the main verb are not ordered. This tree corresponds to the six possible permutations of the three right dependents.

Word order constraints in the elementary trees are expressed by specifying the positional features of some nodes. More precisely, the positional features are defined for all the dependents of the anchor as well as for the anchor when its passive valency is defined in the elementary tree. The three elementary trees of Figure 1, enriched with positional features, are represented in Figure 6. The positional features of the adjective *big*, for example, indicates that it must appear at the left of its governor and that
It is important to note that the `separ` feature of an elementary tree node can refer to siblings that are not present in the elementary tree but which could be added during a tree combining operation, as described in 3.2.



node	side	separ
il	l	*
gaiement	r	*, dir-o, indir-o
ame	r	*, indir-o, circ
son	l	*
à	r	*, dir-o, circ
diable	r	*
le	l	*

Figure 5: A partially ordered dependency tree

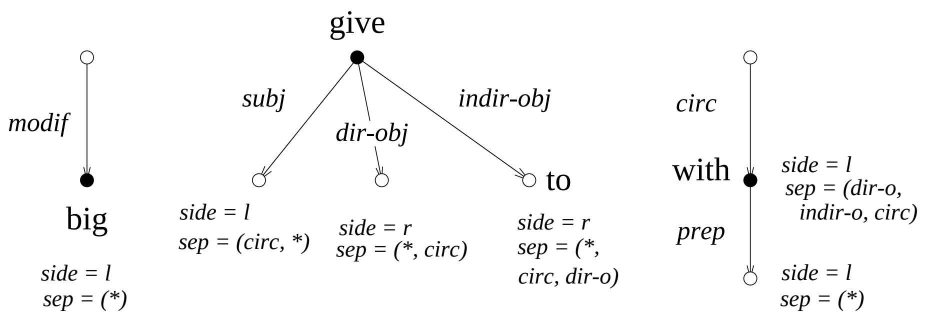


Figure 6: Elementary trees with positional constraints

3.2 Combining elementary trees

Elementary trees constitute the static part of the generative formalism. The dynamic part consists in a single tree combining operation called *attachment*. Contrary to most generative formalisms, the one presented here is not based on rewriting: it is neither a string rewriting system, where a symbol is rewritten by a sequences of other symbols, as in a syntagmatic grammar ([Cho57]), nor a tree rewriting system, where a node is rewritten as a tree, as in Tree Adjoining Grammars. The attachment operation combines two trees to form a new one.

Although this difference is not fundamental, it gives a different status to the two usual products of a derivation: the string and the syntactic tree. In the formalism described here, the syntactic structure is not a by-product of a derivation, as in a string rewriting system; it is the main and actually the unique product of a derivation. The string itself cannot exist without its associated syntactic tree.

Attachment is a non commutative binary operation based on unification. Attaching a tree T_1 *in* a tree T_2 ($Attach(T_1, T_2)$) amounts to unifying the root of T_1 and a node of T_2. We have represented in Figure 7 two attachment operations.

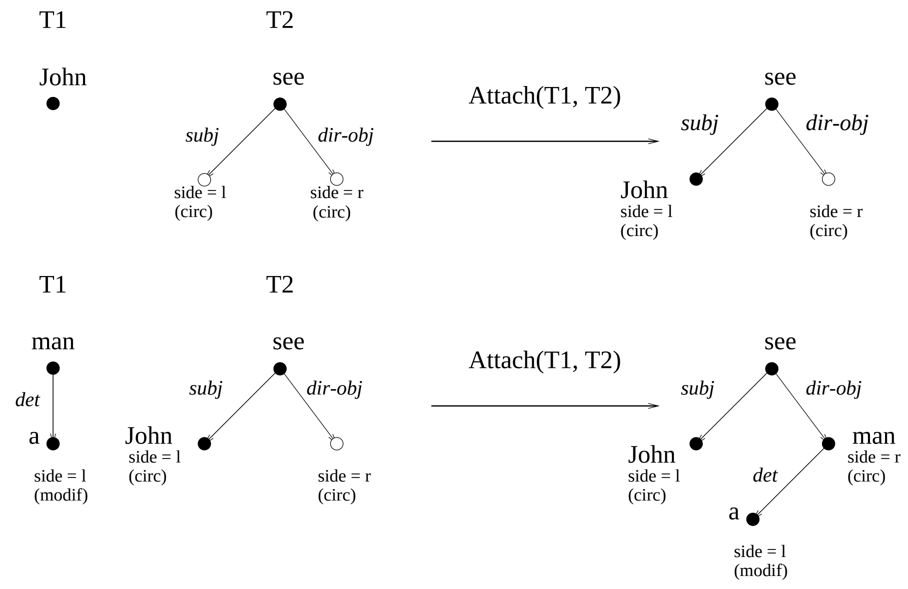


Figure 7: Two attachment operations

Besides the success of the unification, a successful attachment operation must fulfill two more conditions:

Connectivity. After an attachment has been performed and a tree T was produced, the subgraph composed of dependencies linking black nodes of T, which will be called the *black domain of* T, must be connected. As a consequence, during an attachment of a tree T_1 in a tree T_2, a black node of T_1 and a black node of T_2 must be linked together by a dependency. This dependency is called the *attachment dependency* of a particular attachment operation. A schematic attachment operation is represented in Figure 8, which shows the attachment dependency before and after the attachment, in the tree as well as in the corresponding string.

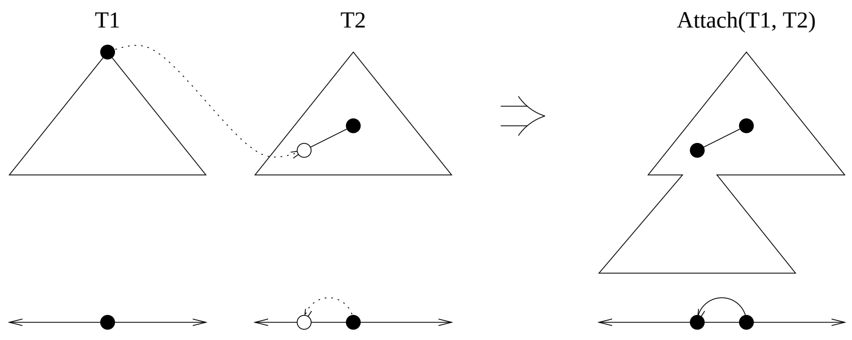


Figure 8: The attachment dependency of an attachment operation

Positional constraints satisfaction. During an attachment, the position of the dependent d of the attachment dependency towards its governor g and towards its same-side siblings must be defined. The `side` feature of d must be set to `left` or `right` or left unspecified. Besides, the `separ` feature of all the dependents of g situated on d side should be compatible, i.e. there exists at least one ordering of these dependents which satisfies all these constraints. If at the conclusion of the operation $Attach(T_1, T_2)$, the strings (there might be several) corresponding to T_1 are situated to the left of the strings corresponding to T_2, the attachment will be called a *left attachment.* If they are situated to the right, the attachment will be called a *right attachment.* If their relative positions is not defined, the attachment will be called a *non directed attachment.* The first example of Figure 7 is therefore a left attachment and the second one, a right attachment.

3.3 The generative procedure

A grammar G is a set of elementary trees. The generative procedure is the following: choose an elementary tree from G and call it T_0. Choose another elementary tree t and attach T_0 in t or t in T_0 to form T_1. The attachment can either be a right, a left or a non directed attachment. Then recursively choose another elementary tree t from G and attach it in T_i or attach T_i in t, to produce T_{i+1}. The generation can stop whenever T_i does not contain any white node. The result of the generation is the tree T_i. This tree represents the syntactic structure of the different node sequences (or strings) which are compatible with the positional features of the nodes. A single derivation can therefore produce several strings.

An example of derivation is represented in Figure 9; it constitutes one of the several possible derivations leading to the final tree. The attachment dependency of every attachment operation is represented in boldface. The different steps of the generation of the string $cbdagef$ are represented below[4]:

$$b \Rightarrow cb \Rightarrow cba \Rightarrow cbda \Rightarrow cbdae \Rightarrow cbdaef \Rightarrow cbdagef$$

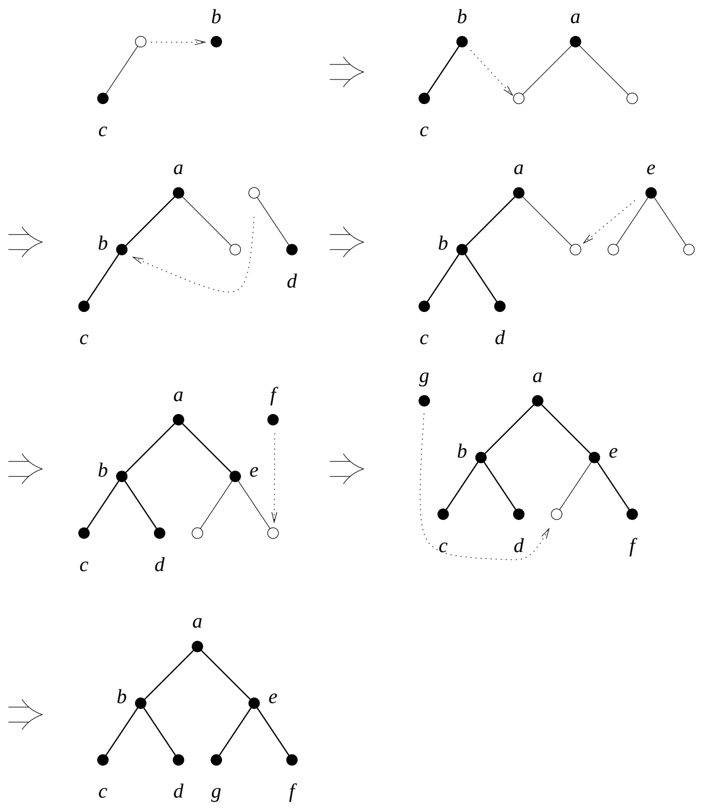


Figure 9: An example of derivation

Note that during an attachment, a symbol can be inserted inside one of the strings being produced ($cba \Rightarrow cbda$). We will call such an attachment an *insertion*. Note that an insertion does not correspond to a left nor to a right attachment. A derivation having no insertions will be called a *contiguous derivation*. We will therefore be interested only in contiguous derivations. Non contiguous derivations have been introduced here for the sole purpose of generality.

Let us study further this kind of derivation since it is the only kind of derivation we can actually perform with our definition of attachment. Contiguous

[4]The reader should not be fooled by such a string representation of the derivation: it is only a compact and incomplete way of representing the stages of a derivation and it is by no means the real derivation. The latter manipulates only trees, as shown in Figure 9.

derivation cannot always be performed in a sequential schema. The generation of the tree of Figure 9, for example, cannot be carried out sequentially. A detailed description of the derivation will illustrate the problem. The subtree corresponding to the sequence *cbda* can be generated contiguously[5]:

$$b \overset{l}{\Rightarrow} cb \overset{l}{\Rightarrow} cbd \overset{r}{\Rightarrow} cbda$$

At this point, g cannot be attached to *cbda* since it depends on e which has not been introduced yet. If e is attached at this point, as in the example of Figure 9, g will have to be inserted. The problem can be avoided by adopting an *asynchronous* derivation schema. The idea is to temporarily stop the generation after *cbda* has been generated [6] and start a new derivation which will lead to the string gef. The tree corresponding to this string is then attached in the *cbda* tree to constitute the final tree. Such stops in the derivation will be called *breaks*. An asynchronous contiguous derivation of our example has been represented below in which breaks are indicated by vertical lines.

$$\begin{array}{r|c|l} b \overset{l}{\Rightarrow} cb \overset{l}{\Rightarrow} cbd \overset{r}{\Rightarrow} cbda & & \overset{l}{\Rightarrow} cbdagef \\ & g \overset{l}{\Rightarrow} ge \overset{r}{\Rightarrow} gef & \end{array}$$

A final type of derivation will be defined now, called *left longest contiguous derivation*, or LLC-derivation. It is a contiguous derivation in which the string is built from left to right and at any moment the leftmost subtree should span the longest possible part of the string. An LLC-derivation of the preceding example have been represented below. Note that, contrary to the previous derivation, the subtree corresponding to *ge* is integrated in *cbda* before the attachment of f in order to satisfy the left longest condition.

$$\begin{array}{r|c|l} c \overset{l}{\Rightarrow} cb \overset{r}{\Rightarrow} cbd \overset{l}{\Rightarrow} cbda & & \overset{r}{\Rightarrow} cbdage \overset{l}{\Rightarrow} cbdagef \\ & g \overset{l}{\Rightarrow} ge & \end{array}$$

The complete tree derivation is represented in Figure 10. The square brackets indicate the beginning and the end of a break.

[5]The letters l and r appearing above the arrows indicate whether a left or a right attachment is performed.

[6]Strictly speaking, we should have written "after the *tree corresponding* to *cbda* has been generated".

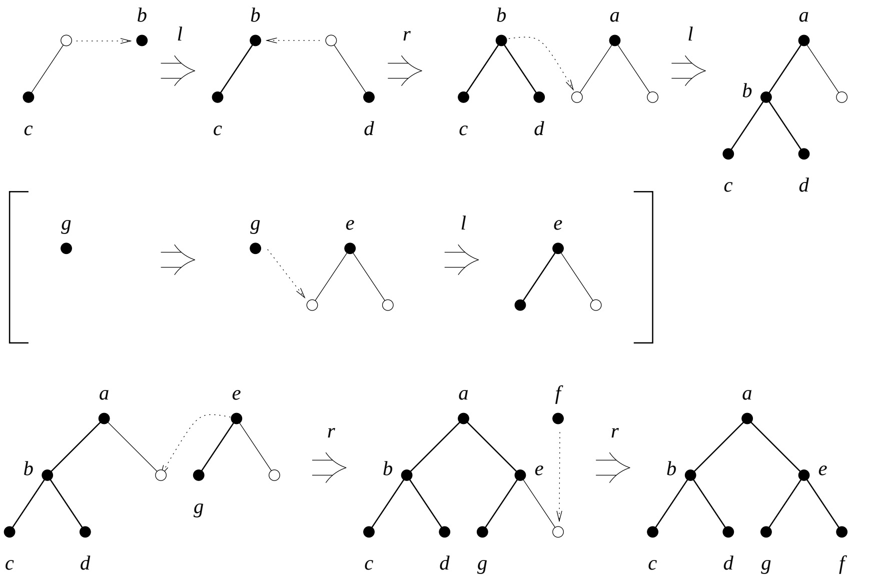


Figure 10: A Left Longest Contiguous derivation

4 Parsing

Having described a generative formalism, we can now define parsing: parsing a symbol sequence (or sentence) S amounts to find all valid derivations of dependency trees corresponding to S[7]. The parser described in this section will build only left longest contiguous derivations. A first, simple, algorithm is described in Subsection 4.1. Some improvements to this algorithm are described in Subsections 4.2 and 4.3 in order to take into account non determinism in the parsing process.

4.1 The parsing algorithm

The parsing algorithm makes use of a stack which will determine the attachment operations to test as well as the order in which to test them. The stack will also keep track of intermediate results when breaks occur. In order to simplify the description of the algorithm, we make the hypothesis of the non ambiguity of the sentence to parse as well as the non ambiguity of the words composing the sentence. As a result, the sentence under analysis will receive a single

[7]This definition of parsing is slightly different from the definition of parsing in a string rewriting framework since in the latter, the result of the derivations which are determined during parsing is the actual string, whereas in our case, as already mentioned, the result of the derivations are dependency trees which are the syntactic representation of the parsed string.

SSyntS and every word of the sentence will be associated to a unique elementary tree. Moreover, during parsing, the algorithm will never be exposed to local ambiguity. Due to these hypothesis, the algorithm will exhibit a deterministic behavior: at any moment of parsing, a single operation can be performed. These restrictions are, of course, absurd, and we will lift them in Subsection 4.2.

Two stack manipulation operations are defined: the standard *push* operation consisting in adding an elementary tree on the top of the stack and a *reduce* operation which consist in trying the attachment of the two trees located at the top of the stack. Two attachment are performed, an attachment of the top element of the stack (T_{TOP}) in its predecessor (T_{PRED}) and the attachment of T_{PRED} in T_{TOP}. The first operation ($Attach(T_{TOP}, T_{PRED})$) must be a right or a non directed attachment since the string corresponding to T_{TOP} is situated to the right of the string corresponding to T_{PRED}. For the same reason, the second operation ($Attach(T_{PRED}, T_{TOP})$) must be a left or a non directed attachment. If one attachment is successful, the two trees are removed from the stack and the resulting tree is pushed on the stack. At the end of the operation, the height of the stack is reduced by one. The push and reduce operations are represented graphically in Figure 11[8].

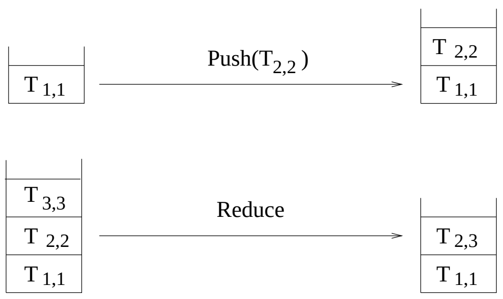


Figure 11: The push and reduce operations

The parsing algorithm consists in reading the sentence from left to right. For every word read, its[9] elementary tree is pushed on the stack and a reduce operation is performed. If the latter is successful, a new tree is created and a new reduce operation is attempted; the process continues until the first unsuccessful reduce operation happens or only one tree remains in the stack. A new word is then read and its elementary tree pushed on the stack, until the last word is reached. If at the end of the process the stack contains a single tree, the parsing

[8]The tree subscripts indicate the segment of the sentence corresponding to the tree. If the elements of the sentence are numbered from 1 to n, tree $T_{i,j}$ corresponds to the segment spanning from symbol i to symbol j, inclusive. Trees of the form $T_{i,i}$ correspond to elementary trees.

[9]recall that we are working under the hypothesis that a word is associated to a single elementary tree

is successful and the tree constitutes the SSyntS of the sentence. The algorithm is represented in Figure 12 and an example of parsing in Figure 13.

```
Input: A list of N elementary trees: $ET_1 \ldots ET_N$
       A stack S

1) For i = 1 to N
do
    2) push($ET_i$)
    3) h $\longleftarrow$ height(S)
    4) Reduce(S)
    5) If height(S) $<$ h Then 3
end do
6) If height(S) $=$ 1 Then Success
                           Else Failure
```

Figure 12: The parsing algorithm

The derivation built during parsing is represented in the linear format below:

$$\mathit{the} \left| \begin{array}{c} \\ \mathit{tall} \overset{l}{\Rightarrow} \mathit{tall\ man} \end{array} \right| \overset{l}{\Rightarrow} \mathit{the\ tall\ man} \overset{l}{\Rightarrow} \mathit{the\ tall\ man\ ran}$$

Note that the break appearing in this derivation is not due to the contiguity condition, as it was the case for the example of Subsection 3.3. It is due to the connectivity condition of the attachment operation. The tree corresponding to the segment *the tall* cannot be created because it would violate the connectivity condition, hence the break.

4.2 Ambiguity and non determinism

As a result of the two hypotheses introduced in 4.1, the parsing algorithm considered is deterministic. After a push operation is performed, a reduce operation is attempted. If the latter succeeds, a new reduce operation is performed and if it fails a push operation is carried out. Taking ambiguity into account will introduce non determinism in the behavior of the parser. Two kinds of ambiguity can occur: *lexical ambiguity*, when a word corresponds to more than one elementary tree (the problem is then to decide which elementary tree is the right one and should be pushed on the stack); and *attachment ambiguity*, when more than one tree are produced after an attachment operation (the problem is then to decide which tree to keep). It must be emphasized that these cases of non determinism cannot be solved locally. It is after the whole sentence or a sufficient part of it has been parsed that we can say whether the chosen hypothesis was the right one. Furthermore, some sentences are ambiguous, in which case,

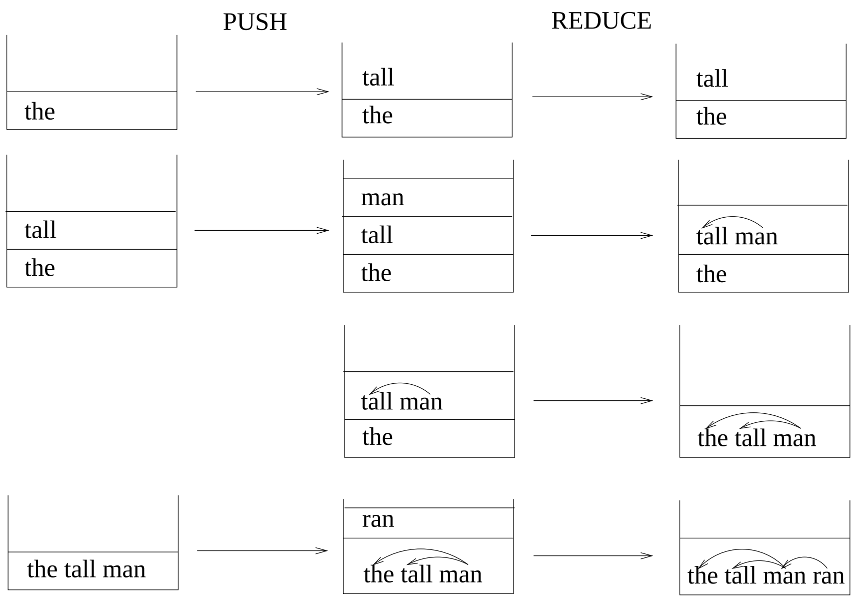


Figure 13: An example of parsing

all the corresponding SSyntS must be built.

4.2.1 Stack duplication

The simplest solution to the two non deterministic situations discussed above consists in the duplication of the stack when the parser is faced with non determinism: each of the new stacks created corresponds to one choice. In the case of a lexical ambiguity where several elementary trees correspond to one word, the stack is duplicated as many times as there are elementary trees and each elementary tree is pushed on one stack. In the case of attachment ambiguity, when an attachment gives rise to more than one tree, the stack is duplicated as many times as there are trees produced and each tree is pushed on one stack. These two cases have been represented in Figure 14.

With a few adjustments of the push and reduce operations in order to take into account several stacks, the rest of the algorithm remains unchanged.

It is easy to see that the time complexity of this algorithm is exponential with respect to the number of words processed since after a word w is read, the number of stacks is multiplied by the number of elementary trees associated to w and a stack is never destroyed during parsing. If N is the mean number of elementary trees associated to a word in the grammar, the mean number of

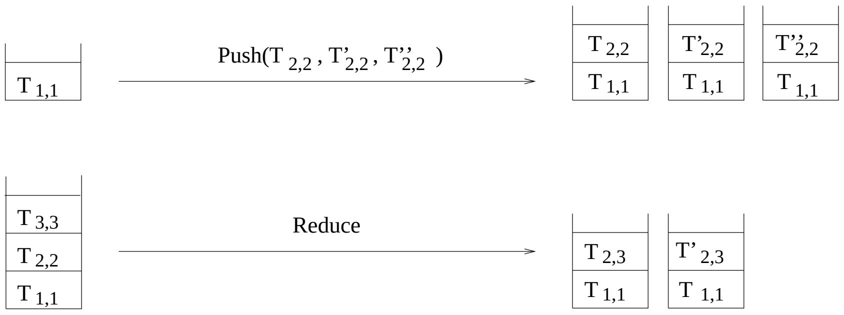


Figure 14: Stack duplication

stacks after n words are read is equal to N^n.[10]

This way of handling ambiguity is not satisfactory since a same attachment can be tested independently several times. Such a configuration is illustrated in Figure 15. In this example, two elementary trees, $T_{1,1}$ and $T'_{1,1}$ are pushed on an empty stack, leading to the duplication of the stack. Another elementary tree, $T_{2,2}$, is then pushed on the two stacks. A reduction (not represented graphically) is then tested on each stack and both fail. Eventually, another elementary tree, $T_{3,3}$, is pushed. At this point, both stacks have the same second and third elements, which means that the next reduction in both stacks will be performed on the same trees.

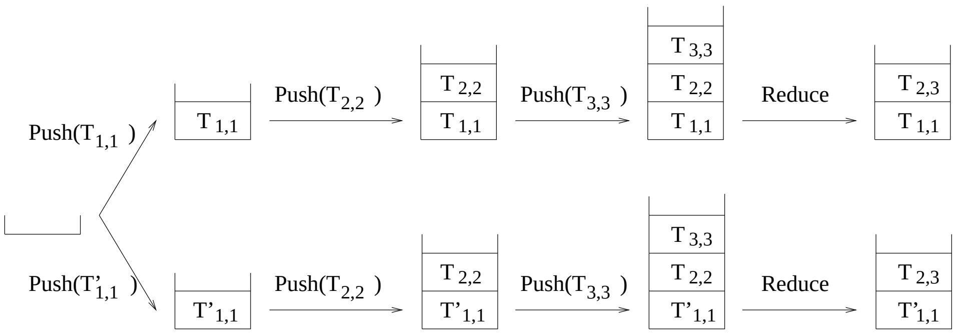


Figure 15: Same operations repeated twice

This problem is due to the fact that the same trees are represented several times in separate stacks and, during reduction, each stack is reduced independently,

[10]Actually, the number of stacks can be higher since the reduction of a stack can also increase the number of stacks.

without considering what has been done in the other stacks. In practice this duplication of the same reduce operations is quite common since, usually, when several elementary trees are associated to a word of the sentence to be parsed, a few number of them will successfully be attached, the other will be stacked without being attached, leading to the duplication of the same configurations in several stacks. We will introduce in the following section a way of partially solving this problem.

4.2.2 Using a Graph-structured Stack

In order to prevent the same work from being done several times in some cases, we use the graph-structured stack introduced in [Tom88]. A graph-structured stack can be seen as a factoring of several stacks having some elements in common, as shown in Figure 16.

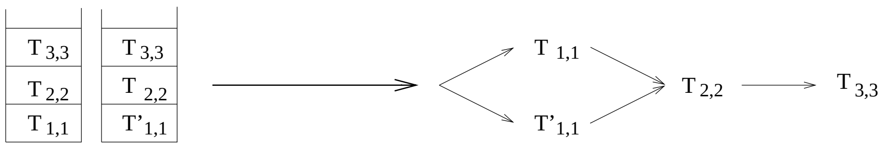


Figure 16: Factoring two stacks into a graph-structured stack

With such a device, any tree is represented once and the attachment of identical pairs of trees is tried once. The introduction of the graph-structured stack does not change the overall schema of the algorithm. Changes affect only the push and the reduction operations. Pushing N elementary trees on a graph-structured stack does not duplicate anymore the stack but adds to it N new extremities and creates a link between every former extremity and every new one, as shown in Figure 17.

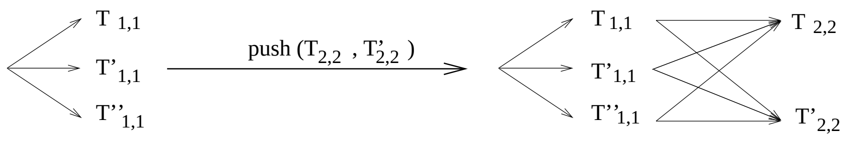


Figure 17: Pushing two elementary trees on a graph-structured stack

The reduction operation amounts to trying to combine each extremity of the stack with all its predecessors. When an attachment operation between an extremity of the stack and one of its predecessors succeeds, new extremities are created and the number of extremities of the stack is therefore increased. In the example of Figure 18 the attachment of extremity $T_{j+1,j+1}$ and its predecessor

$T_{i,j}$ produces two trees: $T_{i,j+1}$ and $T'_{i,j+1}$, increasing by one the number of extremities.

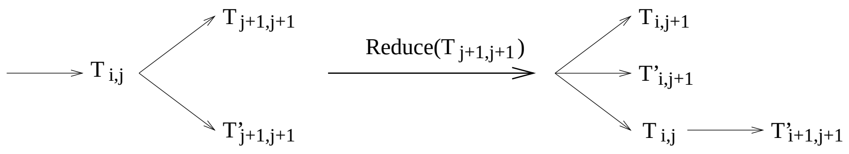


Figure 18: Reducing an extremity of the graph-based stack

The repetition of the same attachment operations that occurred in Figure 15 will not happen with the graph-structured stack, since $T_{2,2}$ and $T_{3,3}$ are represented only once, as shown in Figure 19.

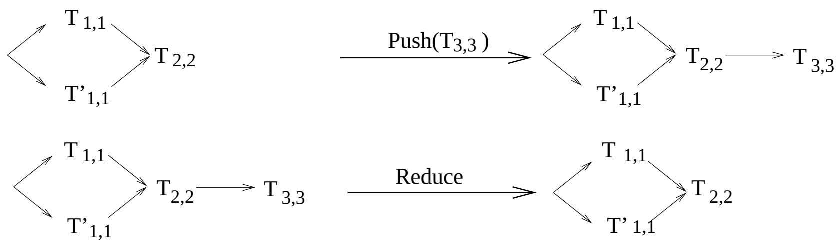


Figure 19: Avoiding repetition of similar operations

The graph-structured stack prevents the attachment of identical trees in different stacks but it does not prevent some work duplication in different branches of the stack. Such a case happens in ambiguous prepositional phrase attachment. Consider the syntactically ambiguous sentence *John saw the man with the telescope that was offered to him last Christmas*. The two syntactic trees corresponding to this sentence differ only in one dependency: the dependency linking *with* to *saw* in one interpretation and to *man* in the other. When parsing such a sentence, the parser will build two different trees after the preposition has been read and the attachment ambiguity has been recognized (*John saw the man with*). Although the analysis of the noun phrase *the telescope that was offered to him last Christmas* is the same in the two cases, it will be done almost[11] twice. There are two ways in which this problem could be solved: the first is to delay the attachment of the noun phrase until it has been completely parsed. The second is to represent in a same structure the two analyses of the sentence.

[11]The attachments corresponding to left dependencies (*the* $\leftarrow$ *telescope*, *last* $\leftarrow$ *Christmas*) are performed only once.

The adoption of a graph-structured stack improves the mean complexity of the parser without changing its worst case complexity. This improvement is mainly due to the fact that, in average, the number of extremities of the graph-structured stack does not grow exponentially with the number of words processed, and hence the number of attachment operations carried out. The worst case complexity remains exponential. It corresponds to the highly improbable case where at each reduction of the graph-structured stack, all the attachment operations performed succeed. In such a case, the number of extremities of the graph-structured stack grows exponentially with the number of word processed.

4.3 Bi-directional parsing

The way non determinism has been handled, either through stack duplication or through the introduction of a graph-structured stack does not always allow to build all the analyses of a given string. In some cases, where several analyses are possible, one solution only will be discovered. Such a case arise, for example, when parsing a sequence $g_1 w g_2$ where the word w can be governed in one hypothesis by g_1 and in another by g_2. In this case, the first hypothesis only, in which g_1 governs w will be built.

A detailed description of the parsing process will help us understand the problem: after g_1 and w have been introduced in the stack, the attachment of w in g_1 is performed [12], leading to the establishment of the right dependency $g_1 \rightarrow w$, preventing the establishment of the left dependency $w \leftarrow g_2$ when g_2 is read and introduced in the stack, since w already has a governor.

This problem does not arise when both possible governors are situated to the left of w, as in the case of prepositional attachment. In such cases, both solutions are built. The problem arise in the cases where w is situated between its possible governors or where the governors are situated to the right of w. Three different types of ambiguity can therefore be distinguished:

- Left ambiguity: the possible governors of w are situated to its left. In this case all the analyses are built.
- Right ambiguity: the possible governors of w are situated to its right. In this case, the analysis in which w is linked to its closest potential governor is built. The other analyses are missed.
- Bilateral ambiguity: the possible governors of w are situated on both sides. In this case, the analyses in which w is linked to its left possible governors are built. The other analyses are missed.

This problem of missed analyses is due to the impatient behavior of the parser which tries to perform an attachment as soon as possible. When the attachment succeeds, the situation that prevailed before the attachment is forgotten,

[12] Strictly speaking, the objects that are actually manipulated are the trees corresponding to w and g_1

possibly preventing the construction of other analyses. It is difficult to predict if a given grammar will give rise to this kind of ambiguities since they might correspond to local ambiguities that appear at a certain stage of the parsing and are ruled out at a later stage. In practice, we have observed such cases with coordinate structures.

Note that the parser exhibit an asymmetrical behavior. When a word w has several possible governors, the parser systematically favors right attachments (where w is linked to its left governors). This behavior is a consequence of the left-to-right direction of parsing: when an elementary tree corresponding to a word w is introduced in the stack, the parser tries to attach it with a right attachment in the part of the sentence which has already been parsed i.e. the part of the sentence situated to the left of w. If the attachment fails, a new elementary tree is introduced and a left attachment of the elementary tree of w in the new elementary tree is performed. The latter is therefore carried out only if the former (the right attachment) fails.

Symmetrically, if the parser processes the sentence from right to left, priority will be given to left attachment: when an elementary tree corresponding to a word w is introduced in the stack, the parser tries to attach it with a left attachment in the part of the sentence which has already been parsed i.e. the part of the sentence situated to the right of w. If the attachment fails, a new elementary tree is introduced and a right attachment is performed. The right attachment is therefore carried out only if the the left attachment fails. Hence, when the parsing is carried out from right to left, in the case of right ambiguity, all hypotheses are detected, in the case of left ambiguity, one hypothesis is detected and in the case of bilateral ambiguities, only hypotheses corresponding to left attachment of w are detected. The situation is summarized in Figure 20 which shows the analyses that are detected for a given type of ambiguity depending on the direction of parsing.

This asymmetrical behavior shows the way to an ad-hoc solution to the missed analyses problem, through a two stages bilateral parsing: in a first stage the sentence is parsed from left to right and in the second stage, it is parsed from right to left. At the end of parsing, some trees are created twice, once during the first stage and then during the second stage. One of them must therefore be discarded. As shown in Figure 20, this case concerns right and left ambiguities.

Changing the parser to process the sentence from right to left is straightforward: it consists in changing the attachments in the reduction operations. The attachment of the tree situated at the top of the stack (T_{TOP}) in its predecessor (T_{PRED}) had to be a right or a non directed attachment. It will now have to be a left or a non directed attachment. Symmetrically the attachment of T_{PRED} in T_{TOP} will now have to be a right or a non directed attachment. Eventually, the sentence has to be traversed from right to left.

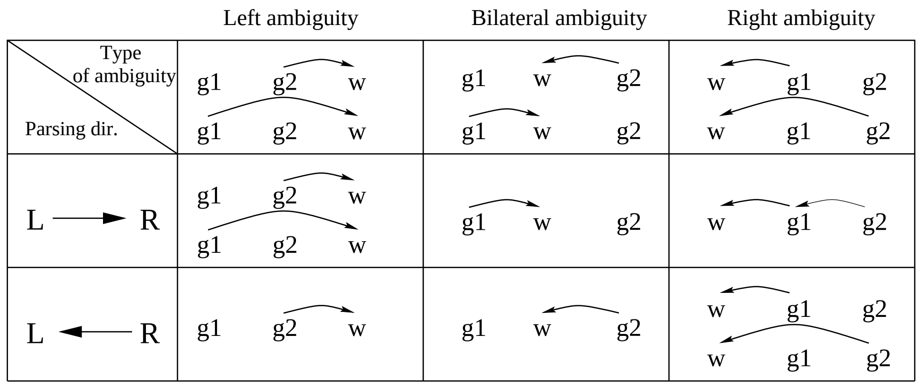


Figure 20: Types of ambiguities detected with respect to the direction of parsing

5 Conclusion

We have proposed in this paper a generative approach to parsing in the framework of the MTT. More precisely, the paper describes a generative formalism and a parser based on this formalism. The reason for proposing an alternate representation of the linguistic knowledge lies in the procedural problems encountered when parsing with a standard MTM.

The proposed formalism is designed to make processing, and parsing in particular, easier. The price to pay for this is an important redundancy of the grammar in the new format. As a consequence, writing a grammar in this format is time consuming and non satisfactory for a single phenomena is described many times in the grammar. Besides, and for the same reason of redundancy, the task of maintaining a grammar is tedious.

The situation could be informally summarized in the following way: “what is good for the linguist is not good for the machine and vice versa”. Two paths can be followed to solve this problem. The first is tradeoff: designing a formalism which is reasonably good (or reasonably bad) for the linguist and for the machine. The second solution is to devise a compiler which takes as input a standard MTM and outputs a grammar in the compiled format. The second solution is to our eyes much more satisfactory. But the compiling process is still unknown to us.

Automating the compiling of a standard MTM has another interesting consequence: it will clarify the relation existing between the piece of work described in this paper and the MTT. Such a relation is, at this point, unclear and the proposed formalism might appear closer to Tree Adjoining Grammars, for example, than to the MTT.

References

[ART93] Margarita Alonso Ramos and Agnès Tutin. Les fonctions lexicales du dictionnaire explicatif combinatoire pour l'étude de la cohésion lexicale. *Linguisticae Investigationes*, XVII(1):161–188, 1993.

[Cho57] Noam Chomsky. *Syntactic Structures.* Mouton & Co, La Haye, 1957.

[Gai65] Haim Gaifman. Dependency systems and phrase-structure systems. *Information and Control*, 8:304–337, 1965.

[GM75] A Gladkij and Igor A. Mel'čuk. Tree grammars I. a formalism for syntactic transformations in natural languages. *Linguistics*, 150:47–82, 1975.

[Hay64] David G Hays. Dependency theory: A formalism and some observations. *Language*, 40:511–525, 1964.

[Hel85] Peter Hellwig. PLAIN: Program for language analysis and inference. Technical report, Computing Unit and Linguistics and International Studies Department, University of Surrey, Guilford, Angleterre, 1985.

[Hel86] Peter Hellwig. Dependency unification grammar. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING'86)*, Bonn, 1986.

[IM99] Lidija Iordanskaja and Igor Mel'čuk. Towards establishing an inventory of surface-syntactic relations. in this volume, 1999.

[JLT75] Aravind Joshi, Leon Levy, and M Takahashi. Tree adjunct grammars. *J. Comput. Syst. Sci.*, 10:136–163, 1975.

[Jos87] Aravind K. Joshi. An introduction to tree adjoining grammars. In A. Manaster-Ramer, editor, *Mathematics of Language*, pages 87–115. John Benjamins, Amsterdam, 1987.

[Kar95] Fred Karlsson. *Constraint Grammar : a language Independent system for Parsing Unrestricted text.* Mouton de Gruyter, 1995.

[KNR98] Sylvain Kahane, Alexis Nasr, and Owen Rambow. Pseudo projectivity: A polynomially parsable non-projective dependency grammar. In *33rd Meeting of the Association for Computational Linguistics (ACL'98)*, Montréal Canada, 1998.

[Lec60] Yves Lecerf. Programme des conflits, modèle des conflits. *Bulletin bimestriel de l'*ATALA, 4,5, 1960.

[Mel88] Igor A. Mel'čuk. *Dependency Syntax: Theory and Practice.* State University of New York Press, New York, 1988.

[MP87a] Igor A. Mel'čuk and Nikolaj V. Pertsov. *Surface Syntax of English.* John Benjamins, Amsterdam/Philadelphia, 1987.

[MP87b] Igor A. Mel'čuk and Alain Polguère. A formal lexicon in the meaning-text theory. *Computational Linguistics*, 13(3-4):13–54, 1987.

[Nas96] Alexis Nasr. *Un modèle de reformulation automatique fondé sur la Théorie Sens Texte: Application aux langues contrôlées.* PhD thesis, Université Paris 7, 1996.

[Rob70] Jane J. Robinson. Dependency structures and transformational rules. *Language*, 46(2):259–285, 1970.

[Shi86] Stuart Shieber. *An introduction to unification based approaches to grammars.* CSLI Series. Chicago Press, 1986.

[Tom88] Masaru Tomita. Graph structured stack and natural language parsing. In *26th Meeting of the Association for Computational Linguistics (ACL'88)*, Buffalo, NY, 1988.