A FORMALISM AND A PARSER FOR LEXICALISED DEPENDENCY GRAMMARS

Alexis NASR[1]
DEA/IA2 Dassault Aviation
78, quai Marcel Dassault - cedex 300 - 92552 St.Cloud - FRANCE
and
TALANA-Universit Paris 7
nasr@dassault-avion.fr

1 Introduction

This paper describes a formalism and a parser for dependency grammars. The grammars written in this formalism are composed of elementary structures, called elementary trees, that are associated to lexemes. Elementary trees can combine together through an operation called attachment. The parser builds a structural description of a sentence by combining the elementary trees associated to its words. An exponential parser is described, based on a stack. The inefficiency of this parser is studied, and another one, using a graph-structured stack, is proposed. Section 2 describes briefly the grammatical formalism: section 2.1 focuses on the structure of elementary trees, section 2.2 deals with word order: how to represent it in a dependency tree and how to constrain it in elementary trees, and section 2.3 introduces the attachment operation. Section 3 is devoted to parsing. The parser is introduced and its performances are discussed in 3.1. The graph-structured stack and its integration in the parser are described in 3.2.

2 The Formalism

In this section, we introduce the grammatical formalism we shall be using for parsing. The grammars written in this formalism consist of finite sets of dependency trees called *elementary trees*. An elementary tree describes a possible syntactic environment of a lexeme. This lexeme is refered to as the *lexical anchor* or simply *anchor* of the tree. There is therefore no strict distinction, in this framework, between the grammar and the lexicon. This is often the case in grammar formalisms proposed for dependency structures, as in [Hellwig, 1986] [Fraser, 1989] and [Starosta, 1986]. The trees can combine together through an operation called attachment. This operation will be used by the parser to build syntactic description of sentences.

2.1 Elementary Trees

An elementary tree is composed of the dependencies its anchor can be involved in, as governor or dependent when it occurs in a sentence. Such structures could include all the possible dependents and governors of an anchor, as it is done in [Fraser, 1989]. But such a description obliges the author of the grammar to anticipate in an elementary tree of a lexeme all of its dependents, modifiers as well as complements. Moreover, in a grammar composed of such elementary trees, each dependency is represented twice, once in the elementary tree of its governor and once in the elementary tree of its dependent. Another solution would be to represent in an elementary tree only the complements of the anchor while modifiers will have in their own elementary trees a description of their governor. This organisation can be compared with the factoring of recursion and dependencies achieved in Tree Adjoining Grammars [Joshi, 1987]. We have

[1]The author wants to thank Anne Abeill, Farid Cerbah, Corinne Fournier and Owen Rambow for their fruitful comments on earlier versions of this paper.

represented in Figure 1 elementary trees corresponding respectively to a verb, a noun and an adjective. The lexical anchors are represented as black nodes. The labels of the dependencies are taken from [Mel'čuk and Pertsov, 1987].

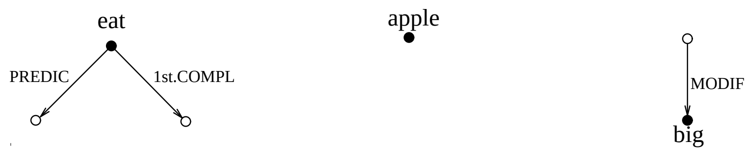


Figure 1: Some elementary trees

Anchors may impose lexical, semantic or morphological restrictions on other words, these words are not always directly linked to the anchor by a syntactic dependency. In order to represent such restrictions in an elementary tree, we introduce these words as nodes in the elementary tree and state lexical, semantic or morphological constraints on them, causing, in some cases, the extension of the domain of locality. Elementary trees can hence have an arbitrarily extended domain of locality and exhibit quite complex patterns, as shown in Figure 2.

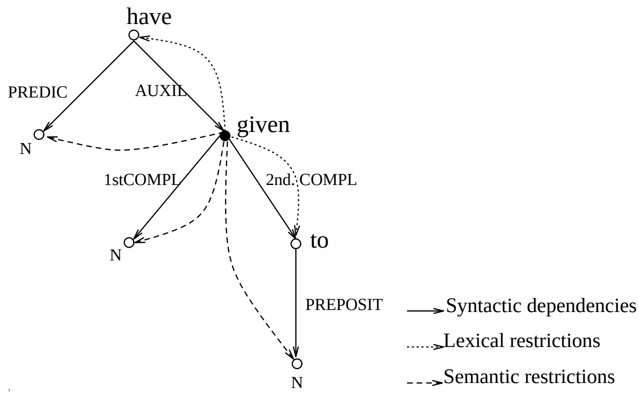


Figure 2: A complex elementary tree

2.1.1 Trees as Complex Feature Structures

The elementary trees are represented as complex feature structures and dependencies as complex attributes. The elementary tree of Figure 2 has been represented in Figure 3 as a complex feature matrix.

$$\left[\begin{array}{l} \text{lex=have} \\ \text{cat=aux} \\ \text{PREDIC=}\left[\begin{array}{l}\text{cat=n}\\ \text{sem=hum}\end{array}\right] \\ \text{AUXIL=}\left[\begin{array}{l} \text{lex=given}\\ \text{cat=past-part}\\ \text{anchor=y}\\ \text{1st.COMPL=}\left[\text{cat=n}\right]\\ \text{2nd.COMPL=}\left[\begin{array}{l}\text{lex=to}\\ \text{cat=prep}\\ \text{PREPOS=}\left[\text{cat=n}\right]\end{array}\right]\end{array}\right]\end{array}\right]$$

Figure 3: An elementary tree as a complex feature matrix

Such structures can be combined together by means of unification. This representational means and the unification operation impose that two attributes of a same structure cannot have the same label, preventing the representation of repeatable dependencies. Several ways to overcome this problem by redefining the unification operation are possible. But we will not develop this point inthis paper.

2.2 Word Order

The status of word order in dependency trees is not as clear as in phrase structure trees and is sometimes not represented at all, as in [Mel'čuk, 1988]. In our grammatical formalism, word order prescriptions, like all grammatical information, is represented in elementary trees. In this section, we propose some notational conventions for representing word order in dependency trees, in such a way that a tree is in correspondence with exactly one linear sequence of its nodes. This notational system will then be used to state constraints on word order in elementary trees.

A dependency tree does not usually define a total order over its nodes: a single tree can correspond to many linear sequences of its nodes. Some enrichments of the trees have been proposed in [Hellwig, 1986] in order to represent linear order. These enrichments amount to indicating, for every node in the tree, its position, in the linear sequence, towards some other nodes of the tree. We shall call this information the *linear coordinates of the node*:

LINEAR COORDINATES OF A NODE

1. its position towards its governor.
2. its position among its sister nodes.

A tree enriched with the linear coordinates of its nodes defines a total order among them, provided that every dependency respects the *projectivity principle*[2].

THE PROJECTIVITY PRINCIPLE

A dependency is projective if its dependent is not separated, in the linear sequence, from the governor by anything apart from descendents of the governor.

We will say indifferently that a dependency or its dependent is projective. A tree made of projective dependencies will be said projective and its linear sequence as well. The projectivity principle drastically limits the number of different linear sequences corresponding to a single dependency tree. We have represented in the left part of Figure 4 a dependency tree and its unique corresponding linear sequence, provided the projectivity principle is enforced. The linear coordinates of the nodes are not represented in the figure explicitly, but implicitly, by the relative horizontal position of the nodes towards their governor and sisters.
Unfortunately, it seems that non projective dependencies cannot be avoided in a grammar and thus must be taken into account. Without the projectivity principle, the knowledge of the linear coordinates of every node of a tree is no longer sufficient to define a total order over them. The tree of Figure 4, for example, enriched with the linear coordinates of its nodes, does not define anymore a total order over its nodes if projectivity is not assumed. We have represented in the right part of Figure 4 the same tree with a non-projective dependency marked with a star and the three linear sequences that now correspond to the tree. In the example, knowing that *an* is situated to the left of its governor (*apple*) is not enough to place it in the linear sequence, it can be placed between *John* and *ate* or to the left of *John*.
We will relax the constraints imposed by the projectivity principle on word order and propose another, less restrictive, principle we will call *pseudo-projectivity* : a projective dependency is

[2]The projectivity principle, also known as the adjacency principle, has been studied by several researchers. The definition given here is due to D.Hudson

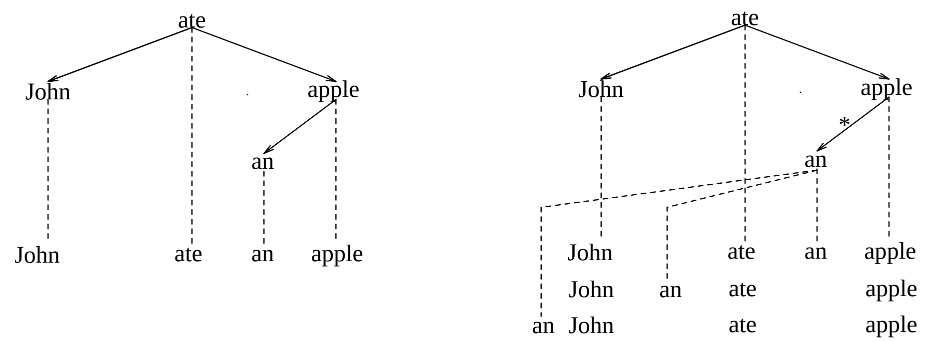


Figure 4: A projective and a non projective tree and their corresponding linear sequences

pseudo-projective but all non projective dependencies are not pseudo-projective.

THE PSEUDO-PROJECTIVITY PRINCIPLE

A dependency is pseudo-projective if its dependent D is not situated, in the linear sequence, between two dependents of a node that is not an ancestor of D.

This principle has been induced from the non projective linguistic examples appearing in [Mel'čuk, 1988] and [Beringer, 1988], all of which satisfy the pseudo-projectivity principle. The three linear sequences of the right part of Figure 4 are pseudo-projective. Contrary to a projective dependency, the dependent D of a pseudo-projective dependency can be separated, in the linear sequence, from its governor by some ancestors of D. We will call the highest of these ancestors the linear governor of D. If the dependency is projective, the linear governor of D is its governor. A node having its governor as linear governor will be said to have a zero non-projectivity level. When the linear governor is the governor's governor, the node will have its non projectivity level equal to one, and so on... We have represented in Figure 5 a tree with a pseudo-projective dependency and the linear sequences corresponding to the tree. To each sequence corresponds a level of non-projectivity of the marked dependency. Triangles hanging from some nodes represent the subtrees rooted by the nodes and the node labels between square brackets represent the linear sequences of the subtrees rooted by the node. By virtue of pseudo-projectivity, the node e cannot be situated in the sequence [a] or [c].

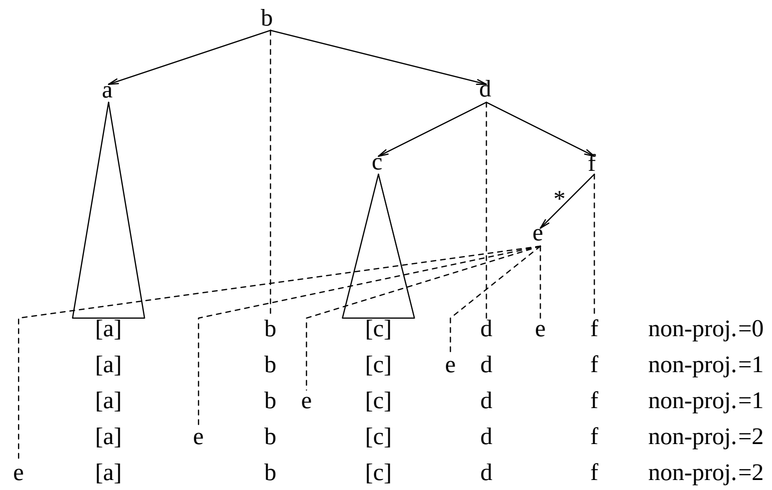


Figure 5: A tree with several non projectivity levels assigned to its non projective dependency

Linear governors will be used to represent the linear order of the nodes of a pseudo-projective tree. We shall define another coordinate system called *pseudo-projective coordinates*.

1. its non projectivity level (or its linear governor).
2. its position towards its linear governor.
3. its position among its linear sisters (dependents of its linear governor).

Enriched with this information, a pseudo-projective tree defines a total order over its nodes. It is interesting to notice that a pseudo-projective tree can be transformed into a projective tree by attaching each node to its linear governor instead of its governor.
This representation of word order will be used to represent word order constraints in elementary trees. But stating such constraints for a node N of an elementary tree might be a problem since they may refer to some nodes (the linear governor and the linear sisters of N) that are outside the domain of locality of the elementary tree. We could decide to extend the domain of locality of the elementary trees and represent the refered node in the domain, as we did for lexical, morphological and semantic restrictions in section 2.1. But such a solution will generate very extended and unnatural elementary trees. Instead we will indicate for some nodes, which of their linear sisters can separate them from their linear governor. More precisely, we shall indicate for the anchor of an elementary tree (when it is not the root of the tree) and for its dependents:

1. their non projectivity level.
2. their relative position towards their linear governor.
3. a list of their linear sisters that can separate them from their linear governor.

The set of these three informations will be refered to as the *positional constraints* of a node.

2.3 Combining Trees

The positional constraints we have defined in the preceding section are not checked during tree combining by unification. We define a tree combining operation we call *tree attachment* that is more specific than unification and takes into account positional constraints. Attaching a tree T1 in a tree T2 amounts to unifying the root of T1 with a node of T2. Two more conditions must be fulfilled by the tree resulting from this operation:

1. The black domain[3] of the resulting tree must be connected (i.e during the attachmentoperation, the black domain of T1 must be linked to the black domain of T2 by a dependency). For example, the attachment of Figure 6 fails because it does not lead to a connected black domain.

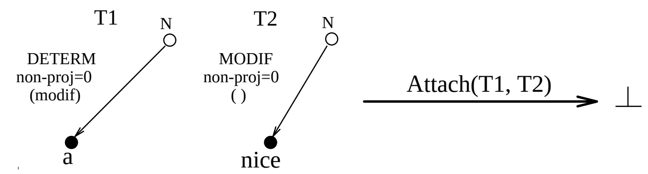


Figure 6: An unsuccessful tree attachment

2. The positional constraints of the dependency linking the black domains of T1 and T2 must be satisfied. Figure 7 shows how the construction of a tree corresponding to the linear sequence "nice a car" fails due to the non satisfaction of node *nice* positional constraints. These constraints prevent the existence of a determiner between a modifier and the modified noun, as indicated by the empty list attached to the dependency MODIF.

[3]The black domain of a tree is the structure composed of all its black nodes and all the dependencies connecting two black nodes.

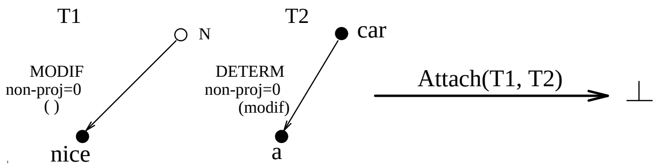


Figure 7: An unsuccessful tree attachment

The attachment of Figure 8 succeeds since the root node of T1 unifies with the node *ate*, the black domain of the resulting tree is connected and positional constraints of the node *with* are satisfied: it is situated to the left of *ate* and separated from it by a 1ST.COMP

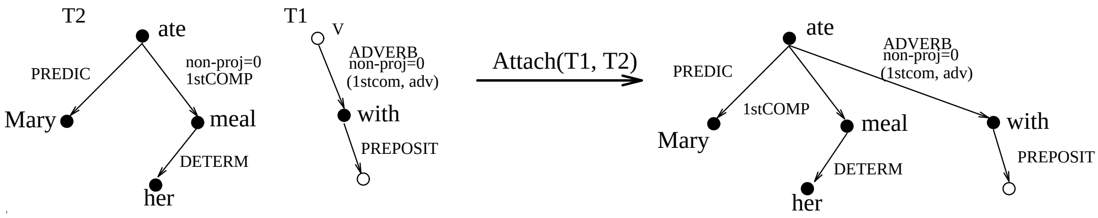


Figure 8: A successful tree attachment

Figure 9 illustrates a successful attachment with a non projective dependency, the node *than* is situated directly to the right of node *salary*, its linear governor.

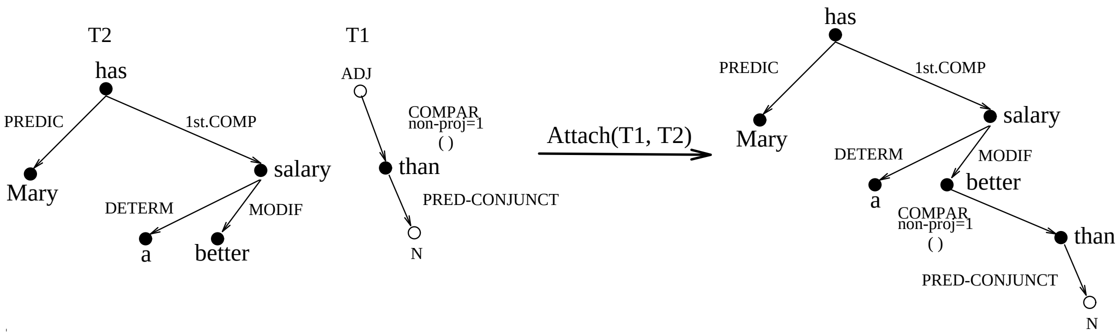


Figure 9: A non projective tree attachment

3 Parsing

In our framework, parsing a sentence amounts to combining, by means of attachment, the elementary trees corresponding to each word of the sentence. The main task of the parser is hence to choose the trees that must be combined together and the order in which they will be combined to build a structural description of a sentence. We describe in this section a first algorithm based on a stack which will turn out to be inefficient. The reasons of this inefficiency are studied and a more efficient parser, that makes use of a graph-structured stack, is proposed.

3.1 A Stack Parser

The usage of a stack in a parser for dependency structures is not a novelty. It has been advocated in [Fraser, 1989] and [Genthial, 1991], mainly to reduce the number of word pairs that must be checked for dependency. The parser processes a sentence from left to right, for each word w_i

read, its elementary trees $T_{i,i}$ [4] are extracted from the lexicon and pushed on the stack. If w_i has N elementary trees, the stack is duplicated N times and each elementary tree is pushed on one stack, as shown in Figure 10.

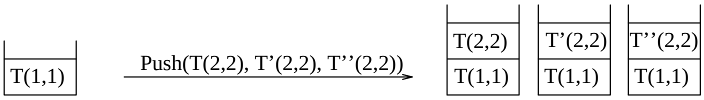


Figure 10: Pushing three elementary trees on a stack

After w_i has been pushed, the N stacks are checked for reduction. The reduction operation tries to combine the top element T1 of the stack with the next one down, T2. The combination is done by the attachment operation. During a stack reduction, two attachments are tested, the attachment of T1 in T2 and the attachment of T2 in T1. If any of these two operations succeeds, the stack height is reduced by one. When the two attachments are successful, the reduction operation gives rise to two trees, in which case, the stack is duplicated, as shown in Figure 11.

Figure 11: Reducing a stack

Then, each stack is checked again for reduction. When the reduction operation fails, the stack is left unchanged, the next word is read and the whole process reiterated. The parsing algorithm stops after the last word w_n has been pushed and the stacks reduced. If at least one stack contains only one element ($T_{1,n}$) the parsing is successful. We have represented in Figure 12 a parse of the non ambiguous sentence "The man ate a red apple". For sake of simplicity we have not represented trees in the stacks but the linear sequences of words. We have also assumed that each word is associated to a single elementary tree, avoiding stack duplication.

3.1.1 Performances of the Stack Parser

The number of stacks during parsing, and hence the number of attachment operations done, grows exponentially with the number of words processed. There are two places where the number of stacks is increased:

1. During the push operation of a word w_i, the number of stacks is multiplied by the number of elementary trees associated to w_i.
2. During the reduction of a stack, when the two attachment operations are successful, the number of stacks is increased.

This parsing algorithm is not satisfactory since a same attachment can be tested independently several times. Such a configuration is illustrated in Figure 13 where the attachments of $T_{2,2}$ and $T_{3,3}$ are tested 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, without any consideration of what has been done in the other stacks.

[4] $T_{i,j}$ is a tree corresponding to the segment of the sentence $[w_i, w_j]$ with $i \leq j$. $T_{x,x}$ is an elementary tree corresponding to w_x.

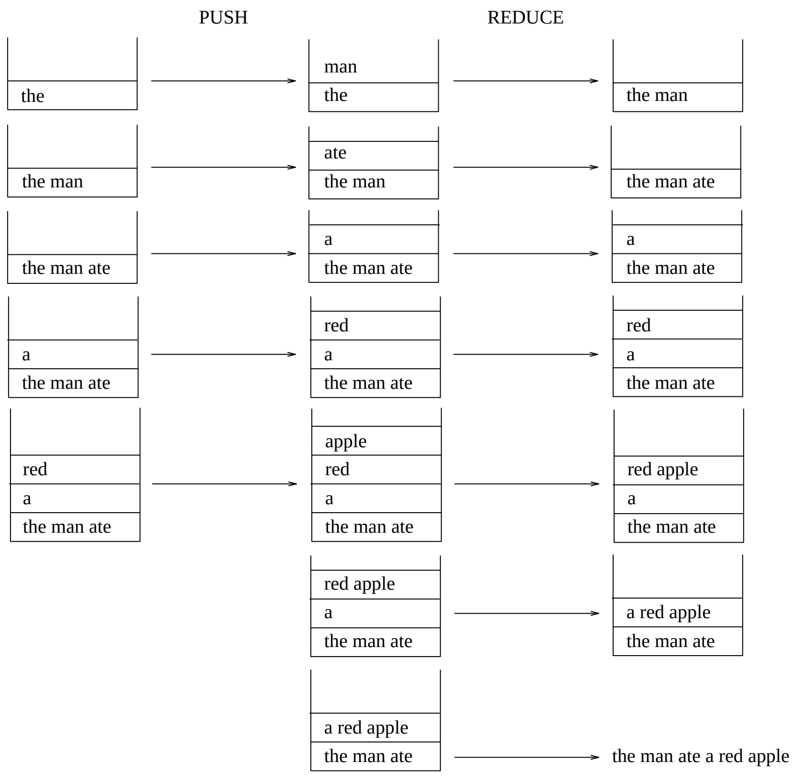
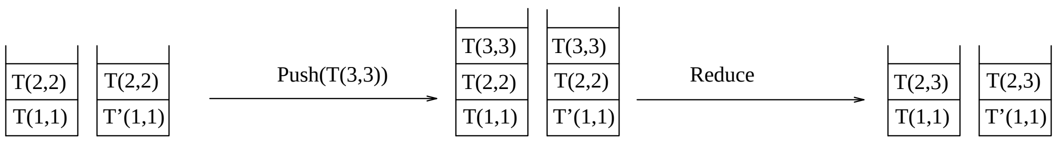


Figure 12: Parsing the sentence "The man ate a red apple"

3.2 Using a Graph-structured Stack

In order to prevent the same work from being done several times, we have used the graph-structured stack described in [Tomita, 1988]. A graph-structured stack can be seen as a factoring of several stacks having some elements in common, as shown in Figure 14.
With such a device, any tree is represented once and the attachment of identical pairs of tree 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 operation.
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 15.
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, the number of extremities of the stack can be increased. In the example of Figure 16 the attachment of extremity $T_{j+1,j+1}$ and its predecessor $T_{i,j}$ produces two trees:



Attachment tested during reduction:

$$2 * Attach(T_{2,2}, T_{3,3}) \; ; \; 2 * Attach(T_{3,3}, T_{2,2})$$

Figure 13: Same operations repeated twice

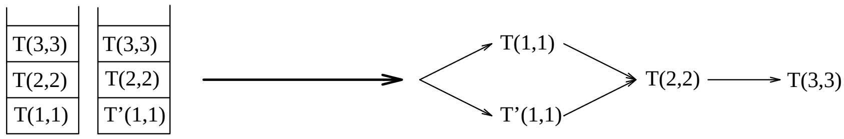


Figure 14: Factoring two stacks into a graph-structured stack

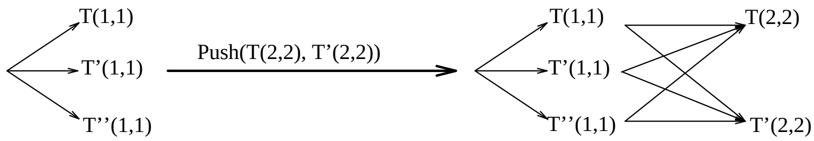


Figure 15: Pushing two elementary trees on a graph-structured stack

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

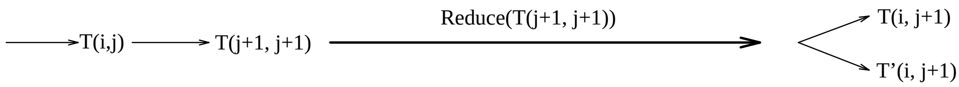
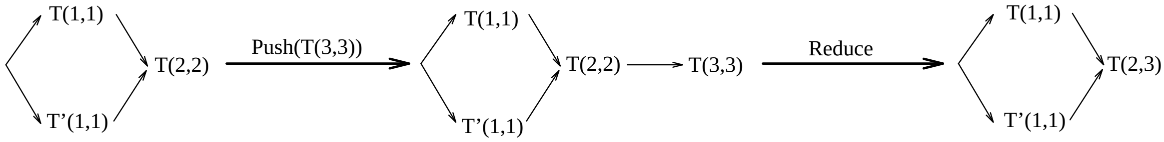


Figure 16: Reducing an extremity of the graph-based stack

The repetition of the same operations that occurred in Figure 13 will not happen with the graph-structured stack, as shown in Figure 17.



Attachment tested during reduction:

$$Attach(T_{2,2}, T_{3,3}) \ ; \ Attach(T_{3,3}, T_{2,2})$$

Figure 17: Avoiding repetition of similar operations

The adoption of a graph-structured stack improves the time complexity of the parser. This improvement is mainly due to the fact that the number of extremities of the graph stack does not grow exponentially with the number of words processed, and hence the number of attachment operations carried out. As predicted by Tomita, the time complexity of the parser is polynomial.

4 Conclusions and Future Work

We have described in this paper a formalism and a parser for dependency grammars, focussing on the representation of word order in dependency trees and the advantages of using a graph-structured stack for parsing. This work is part of a larger project of sentence paraphrasing. Paraphrasing, in this framework, is based on substitution of elementary trees in syntactic description of sentences. After a sentence has been parsed and a dependency tree built, some of the elementary trees forming the dependency tree of the sentence will be replaced by other elementary trees to which they are linked. The modified syntactic description gives rise to a paraphrase of the inital sentence. Future work will concern the relations between elementary trees and the replacement of an elementary tree by another in a larger tree.

References

[Beringer, 1988] Beringer, H. (1988). *Traitement Automatique des Ambiguités Structurales du Français utilisant la Théorie Sens-Texte.* PhD thesis, Université de Paris 6.

[Fraser, 1989] Fraser, N. (1989). Parsing and dependency grammar. *UCL Working Papers in Linguistics*, 1:296–319.

[Genthial, 1991] Genthial, D. (1991). *Contribution à la construction d'un système robuste d'-analyse du français.* PhD thesis, Université Joseph Fourier.

[Hellwig, 1986] Hellwig, P. (1986). Dependency unification grammar. In *coling86*, Bonn.

[Joshi, 1987] Joshi, A. K. (1987). An introduction to Tree Adjoining Grammars. In Manaster-Ramer, A., editor, *Mathematics of Language*, pages 87–115. John Benjamins, Amsterdam.

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

[Mel'čuk and Pertsov, 1987] Mel'čuk, I. A. and Pertsov, N. V. (1987). *Surface Syntax of English.* John Benjamins, Amsterdam/Philadelphia.

[Starosta, 1986] Starosta, S. (1986). Lexicase parsing : A lexicon-driven approach to syntactic analysis. In *Proceedings of the 11th International Conference on Computational Linguistics (COLING'86)*, Bonn.

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