Formal aspects and parsing issues of dependency theory

Vincenzo Lombardo and Leonardo Lesmo
Dipartimento di Informatica and Centro di Scienza Cognitiva
Universita' di Torino
c.so Svizzera 185 - 10149 Torino - Italy
{vincenzo, lesmo}@di.unito.it

Abstract

The paper investigates the problem of providing a formal device for the dependency approach to syntax, and to link it with a parsing model. After reviewing the basic tenets of the paradigm and the few existing mathematical results, we describe a dependency formalism which is able to deal with long-distance dependencies. Finally, we present an Earley-style parser for the formalism and discuss the (polynomial) complexity results.

1. Introduction

Many authors have developed dependency theories that cover cross-linguistically the most significant phenomena of natural language syntax: the approaches range from generative formalisms (Sgall et al. 1986), to lexically-based descriptions (Mel'cuk 1988), to hierarchical organizations of linguistic knowledge (Hudson 1990) (Fraser, Hudson 1992), to constrained categorial grammars (Milward 1994). Also, a number of parsers have been developed for some dependency frameworks (Covington 1990) (Kwon, Yoon 1991) (Sleator, Temperley 1993) (Hahn et al. 1994) (Lombardo, Lesmo 1996), including a stochastic treatment (Eisner 1996) and an object-oriented parallel parsing method (Neuhaus, Hahn 1996).
However, dependency theories have never been explicitly linked to formal models. Parsers and applications usually refer to grammars built around a core of dependency concepts, but there is a great variety in the description of syntactic constraints, from rules that are very similar to CFG productions (Gaifman 1965) to individual binary relations on words or syntactic categories (Covington 1990) (Sleator, Temperley 1993).

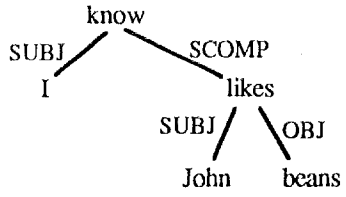

Figure 1. A dependency tree for the sentence "I know John likes beans". The leftward or rightward orientation of the edges represents the order constraints: the dependents that precede (respectively, follow) the head stand on its left (resp. right).

The basic idea of dependency is that the syntactic structure of a sentence is described in terms of binary relations (*dependency relations*) on pairs of words, a *head* (parent), and a *dependent* (daughter), respectively; these relations usually form a tree, the *dependency tree* (fig. 1).

The linguistic merits of dependency syntax have been widely debated (e.g. (Hudson 1990)). Dependency syntax is attractive because of the immediate mapping of dependency trees on the predicate-arguments structure and because of the treatment of free-word order constructs (Sgall et al. 1986) (Mel'cuk 1988). Desirable properties of lexicalized formalisms (Schabes 1990), like finite ambiguity and decidability of string acceptance, intuitively hold for dependency syntax.

On the contrary, the formal studies on dependency theories are rare in the literature. Gaifman (1965) showed that projective dependency grammars, expressed by dependency rules on syntactic categories, are weakly equivalent to context-free grammars. And, in fact, it is possible to devise $O(n^3)$ parsers for this formalism (Lombardo, Lesmo 1996), or other projective variations (Milward 1994) (Eisner 1996). On the controlled relaxation of projective constraints, Nasr (1995) has introduced the condition of pseudo-projectivity, which provides some controlled looser constraints on arc crossing in a dependency tree, and has developed a polynomial parser based on a graph-structured stack. Neuhaus and Broker (1997) have recently showed that the general recognition problem for non-projective dependency grammars (what they call *discontinuous DG*) is NP-complete. They have devised a discontinuous DG with exclusively lexical categories (no traces, as most dependency theories do), and dealing with free word order constructs through a looser subtree ordering. This formalism, considered as the most straightforward extension to a projective formalism, permits the reduction of the vertex cover problem to the dependency recognition problem, thus yielding the NP-completeness result.

However, even if banned from the dependency literature, the use of non lexical categories is only a notational variant of some graph structures already present in some formalisms (see, e.g., Word Grammar (Hudson 1990)). This paper introduces a lexicalized dependency formalism, which deals

with long distance dependencies, and a polynomial parsing algorithm. The formalism is projective, and copes with long-distance dependency phenomena through the introduction of non lexical categories. The non lexical categories allow us to keep inalterate the condition of projectivity, encoded in the notion of derivation. The core of the grammar relies on predicate-argument structures associated with lexical items, where the head is a word and dependents are categories linked by edges labelled with dependency relations. Free word order constructs are dealt with by constraining displacements via a set data structure in the derivation relation. The introduction of non lexical categories also permits the resolution of the inconsistencies pointed out by Neuhaus and Broker in Word Grammar (1997).

The parser is an Earley type parser with a polynomial complexity, that encodes the dependency trees associated with a sentence.

The paper is organized as follows. The next section presents a formal dependency system that describes the linguistic knowledge. Section 3 presents an Earley-type parser: we illustrate the algorithm, trace an example, and discuss the complexity results. Section 4 concludes the paper.

2. A dependency formalism

The basic idea of dependency is that the syntactic structure of a sentence is described in terms of binary relations (*dependency relations*) on pairs of words, a *head* (or parent), and a *dependent* (daughter), respectively; these relations form a tree, the *dependency tree*. In this section we introduce a formal dependency system. The formalism is expressed via dependency rules which describe one level of a dependency tree. Then, we introduce a notion of derivation that allows us to define the language generated by a dependency grammar of this form.

The grammar and the lexicon coincide, since the rules are lexicalized: the head of the rule is a word of a certain category, i.e. the lexical anchor. From the linguistic point of view we can recognize two types of dependency rules: *primitive* dependency rules, which represent subcategorization frames, and *non-primitive* dependency rules, which result from the application of *lexical metarules* to primitive and non-primitive dependency rules. Lexical metarules (not dealt with in this paper) obey general principles of linguistic theories.

A *dependency grammar* is a six-tuple <W, C, S, D, I, H>, where

W is a finite set of symbols (words of a natural language);

C is a set of syntactic categories (among which the special category E);

S is a non-empty set of root categories ($C \supseteq S$);

D is the set of *dependency relations*, e.g. SUBJ, OBJ, XCOMP, P-OBJ, PRED (among which the special relation VISITOR[1]);

I is a finite set of symbols (among which the special symbol 0), called *u-indices*;

H is a set of *dependency rules* of the form

$x{:}X\ (\langle r_1 Y_1 u_1 \tau_1\rangle \ldots \langle r_{i-1} Y_{i-1} u_{i-1} \tau_{i-1}\rangle \# \langle r_{i+1} Y_{i+1} u_{i+1} \tau_{i+1}\rangle \ldots \langle r_m Y_m u_m \tau_m\rangle)$

1) $x \in W$, is the *head* of the rule;
2) $X \in C$, is its syntactic category;
3) an element $\langle r_j\ Y_j\ u_j\ \tau_j\rangle$ is a *d-quadruple* (which describes a dependent); the sequence of d-quads, including the symbol # (representing the linear position of the head, # is a special symbol), is called the *d-quad sequence*. We have that
   3a) $r_j \in D, j \in \{1, \ldots, i-1, i+1, \ldots, m\}$;
   3b) $Y_j \in C, j \in \{1, \ldots, i-1, i+1, \ldots, m\}$;
   3c) $u_j \in I, j \in \{1, \ldots, i-1, i+1, \ldots, m\}$;
   3d) τ_j is a (possibly empty) set of triples <u, r, Y>, called *u-triples*, where $u \in I$, $r \in D$, $Y \in C$.

Finally, it holds that:

I) For each $u \in I$ that appears in a u-triple <u, r, $Y \rangle \in U_j$, there exists exactly one d-quad $\langle r_i Y_i u_i \tau_i\rangle$ in the same rule such that $u = u_i$, $i \neq j$.

II) For each $u = u_i$ of a d-quad $\langle r_i Y_i u_i \tau_i\rangle$, there exists exactly one u-triple <u, r, $Y \rangle \in \tau_j$, $i \neq j$, in the same rule.

Intuitively, a dependency rule constrains one node (head) and its dependents in a dependency tree: the d-quad sequence states the order of elements, both the head (# position) and the dependents (d-quads). The grammar is lexicalized, because each dependency rule has a lexical anchor in its head (x:X). A d-quad $\langle r_j Y_j u_j \tau_j\rangle$ identifies a dependent of category Y_j, connected with the head via a dependency relation r_j. Each element of the d-quad sequence is possibly associated with a u-index (u_j) and a set of u-triples (τ_j). Both u_j and τ_j can be null elements, i.e. 0 and $\emptyset$, respectively. A u-triple (τ-component of the d-quad) <u, R, Y> bounds the area of the dependency tree where the trace can be located. Given the constraints I and II, there is a one-to-one correspondence between the u-indices and the u-triples of the d-quads. Given that a dependency rule constrains one head and its direct dependents in the dependency tree, we have that the dependent indexed by u_k is coindexed with a

[1] The relation VISITOR (Hudson 1990) accounts for displaced elements and, differently from the other relations, is not semantically interpreted.

trace node in the subtree rooted by the dependent containing the u-triple $<u_k, R, Y>$.

Now we introduce a notion of derivation for this formalism. As one dependency rule can be used more than once in a derivation process, it is necessary to replace the u-indices with unique symbols (progressive integers) before the actual use. The replacement must be consistent in the u and the τ components. When all the indices in the rule are replaced, we say that the dependency rule (as well as the u-triple) is instantiated.

A triple consisting of a word w ($\in$ W) or the trace symbol ε ($\notin$ W) and two integers μ and ν is a *word object* of the grammar.

Given a grammar G, the set of word objects of G is

$$W_X(G)=\{ {}_\mu x_\nu \,/\, \mu,\nu\geq 0,\ x\in W\cup\{\varepsilon\}\}.$$

A pair consisting of a category X ($\in$ C) and a string of instantiated u-triples γ is a *category object* of the grammar ($X(\gamma)$).

A 4-tuple consisting of a dependency relation r ($\in$ D), a category object $X(\gamma_1)$, an integer k, a set of instantiated u-triples γ_2 is a *derivation object* of the grammar. Given a grammar G, the set of derivation objects of G is

$C_X(G)$ = {<r,Y(γ_1),u,γ_2> /
r$\in$ D, Y$\in$ C, u is an integer,
γ_1,γ_2 are strings of instantiated u-triples}.

Let $\alpha,\beta\in W_X(G)^*$ and $\psi\in(W_X(G)\cup C_X(G))^*$. The derivation relation holds as follows:

1)

$\alpha <R,X(\gamma_p),u,\gamma_x>\ \psi \Rightarrow\ \alpha$
$<r_1,Y_1(\rho_1),u_1,\tau_1>$
$<r_2,Y_2(\rho_2),u_2,\tau_2>$
...
$<r_{i-1},Y_{i-1}(\rho_{i-1}),u_{i-1},\tau_{i-1}>$
${}_u x_0$
$<r_{i+1},Y_{i+1}(\rho_{i+1}),u_{i+1},\tau_{i+1}>$
...
$<r_m,Y_m(\rho_m),u_m,\tau_m>$
ψ

where $x{:}X\ (<r_1 Y_1 u_1 \tau_1> \ldots <r_{i-1} Y_{i-1} u_{i-1} \tau_{i-1}> \#$ $<r_{i+1} Y_{i+1} u_{i+1} \tau_{i+1}> \ldots <r_m Y_m u_m \tau_m>)$ is a dependency rule, and $\rho_1 \cup \ldots \cup \rho_m = \gamma_p \cup \gamma_x$.

2)

$\alpha <r,X(<j,r,X>),u,\varnothing>\ \psi \Rightarrow\ \alpha\ {}_u\varepsilon_j\ \psi$

We define $\Rightarrow^*$ as the reflexive, transitive closure of $\Rightarrow$.

Given a grammar G, L'(G) is the language of sequences of word objects:

$L'(G)=\{\alpha\in W_X(G)^* /$
$<TOP, Q(\varnothing), 0, \varnothing> \Rightarrow^* \alpha$ and $Q\in S(G)\}$

where TOP is a dummy dependency relation. The language generated by the grammar G, L(G), is defined through the function t:

$L(G)=\{w\in W_X(G)^* / w=t(\alpha)$ and $\alpha\in L'(G)\}$,

where t is defined recursively as

$t(-) = -$;
$t({}_\mu w_\nu\ \alpha) = w\ t(\alpha)$;
$t({}_\mu\varepsilon_\nu\ \alpha) = t(\alpha)$.

where - is the empty sequence.

As an example, consider the grammar

G_1 = <
$W(G_1)$ = {I, John, beans, know, likes}
$C(G_1)$ = {V, V+EX, N}
$S(G_1)$ = {V, V+EX}
$D(G_1)$ = {SUBJ, OBJ, VISITOR, TOP}
$I(G_1)$ = {0, u_1}
$T(G_1)$ >,

where $T(G_1)$ includes the following dependency rules:

1. I: N (#);
2. John: N (#);
3. beans: N (#);
4. likes: V (<SUBJ, N, 0, $\varnothing$> # <OBJ, N, 0, $\varnothing$)>);
5. know: V+EX (<VISITOR, N, u1, $\varnothing$>
<SUBJ, N, 0, $\varnothing$>
#
<SCOMP, V, 0, {<u1,OBJ,N>}>).

A derivation for the sentence "Beans I know John likes" is the following:

<TOP, V+EX($\varnothing$), 0, $\varnothing$> $\Rightarrow$
<VISITOR, N($\varnothing$), 1, $\varnothing$> <SUBJ, N($\varnothing$), 0, $\varnothing$> know
<SCOMP, V($\varnothing$), 0, {<1,OBJ,N>}> $\Rightarrow$
${}_1$beans <SUBJ, N($\varnothing$), 0, $\varnothing$> know
<SCOMP, V($\varnothing$), 0, {<1,OBJ,N>}> $\Rightarrow$
${}_1$beans I know <SCOMP, V($\varnothing$), 0, {<1,OBJ,N>}> $\Rightarrow$
${}_1$beans I know <SUBJ, N($\varnothing$), 0, $\varnothing$> likes
<OBJ, N(<1,OBJ,N>), 0, $\varnothing$> $\Rightarrow$
${}_1$beans I know John likes
<OBJ, N(<1,OBJ,N>), 0, $\varnothing$> $\Rightarrow$
${}_1$beans I know John likes ε_1

The dependency tree corresponding to this derivation is in fig. 2.

3. Parsing issues

In this section we describe an Earley-style parser for the formalism in section 2. The parser is an off-line algorithm: the first step scans the input sentence to select the appropriate dependency rules

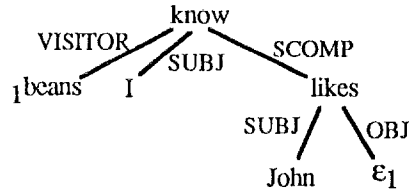

Figure 2. Dependency tree of the sentence "Beans I know John likes", given the grammar G_1.

from the grammar. The selection is carried out by matching the head of the rules with the words of the sentence. The second step follows Earley's phases on the dependency rules, together with the treatment of u-indices and u-triples. This off-line technique is not uncommon in lexicalized grammars, since each Earley's prediction would waste much computation time (a grammar factor) in the body of the algorithm, because dependency rules do not abstract over categories (cf. (Schabes 1990)).

In order to recognize a sentence of n words, n+1 sets S_i of items are built. An item represents a subtree of the total dependency tree associated with the sentence. An item is a 5-tuple <Dotted-rule, Position, μ-index, ν-index, T-stack>. *Dotted-rule* is a dependency rule with a dot between two d-quads of the d-quad sequence. *Position* is the input position where the parsing of the subtree represented by this item began (the leftmost position on the spanned string). μ-index and ν-index are two integers that correspond to the indices of a word object in a derivation. *T-stack* is a stack of sets of u-triples to be satisfied yet: the sets of u-triples (including empty sets, when applicable) provided by the various items are stacked in order to verify the consumption of one u-triple in the appropriate subtree (cf. the notion of derivation above). Each time the parser predicts a dependent, the set of u-triples associated with it is pushed onto the stack. In order for an item to enter the completer phase, the top of *T-stack* must be the empty set, that means that all the u-triples associated with that item have been satisfied. The sets of u-triples in T-stack are always inserted at the top, after having checked that each u-triple is not already present in T-stack (the check neglects the u-index). In case a u-triple is present, the deeper u-triple is deleted, and the T-stack will only contain the u-triple in the top set (see the derivation relation above). When satisfying a u-triple, the T-stack is treated as a set data structure, since the formalism does not pose any constraint on the order of consumption of the u-triples.

Following Earley's style, the general idea is to move the dot rightward, while predicting the dependents of the head of the rule. The dot can advance across a d-quadruple $<r_j Y_j u_j \tau_j>$ or across the special symbol #. The d-quad immediately following the dot can be indexed u_j. This is acknowledged by predicting an item (representing the subtree rooted by that dependent), and inserting a new progressive integer in the fourth component of the item (ν-index). τ_j is pushed onto T-stack: the substructure rooted by a node of category Y_j must contain the trace nodes of the type licensed by the u-triples. The prediction of a trace occurs as in the case 2) of the derivation process. When an item P contains a dotted rule with the dot at its end and a T-stack with the empty set $\varnothing$ as the top symbol, the parser looks for the items that can advance the dot, given the completion of the dotted dependency rule in P. Here is the algorithm.

Sentence: $w_0\ w_1 \ldots w_{n-1}$
Grammar G=<W,C,S,D,I,H>

initialization
for each x:Q(δ)$\in$ H(G), where Q$\in$ S(G)
  replace each u-index in δ with a progressive integer;
  INSERT <x:Q(• δ), 0, 0, 0, []> into S_0
body
for each set S_i (0≤i≤n) do
  for each P=<y: Y(η • δ), j, μ, ν, T-stack> in S_i
---> ***completer*** (including *pseudocompleter*).
  if δ is the empty sequence and TOP(T-stack)=$\varnothing$
    for each <x: X(λ • <R, Y, u_x, τ_x> ζ),
          j', μ', ν', T-stack'> in S_j
      T-stack" <- POP(T-stack);
      INSERT <x: X(λ <R, Y, u_x, τ_x> • ζ),
          j', μ', ν', T-stack"> into S_i;
---> ***predictor***:
  if δ = <R δ, Z δ, u_δ, τ_δ> η then
    for each rule z: Z δ(θ)
      replace each u-index in θ with a prog. integer;
      T-stack' <- PUSH-UNION(τ_δ, T-stack);
      INSERT <z: Z δ(• θ), i, 0, u_δ, T-stack'> into S_i;
----------> *pseudopredictor:*
    if <u, R δ, Z δ> $\in$ UNION (set_i | set_i in T-stack);
      DELETE <u, R δ, Z δ> from T-stack;
      T-stack' <- PUSH($\varnothing$, T-stack);
      INSERT <ε: Z δ(• #), i, u, u δ, T-stack'> into S_i;
---> ***scanner*** :
  if δ = # η then
    if y=w_i
      INSERT <y: Y(γ # • η),
          j, μ, ν, T-stack> into S_{i+1}
----------> *pseudoscanner:*
    elseif y=ε
      INSERT <ε: Y(# •), j, μ, ν, T-stack> into S_i;
  endfor
endfor;
termination
if <x: Q(α •), 0, μ, ν, []> $\in$ S_n, where Q$\in$ S(G)
then accept else reject endif.

At the beginning (*initialization*), the parser initializes the set S_0, by inserting all the dotted rules (x:Q(δ)$\in$ H(G)) that have a head of a root category (Q$\in$ S(G)). The dot precedes the whole d-quad sequence (δ). Each u-index of the rule is replaced by a progressive integer, in both the u and

the τ components of the d-quads. Both μ and ν-indices are null (0), and T-stack is empty ([]).

The body consists of an external loop on the sets S_i ($0 \le i \le n$) and an inner loop on the single items of the set S_i. Let

$$P = \langle y: Y(\eta \bullet \delta), j, \mu, \nu, \text{T-stack}\rangle$$

be a generic item. Following Earley's schema, the parser invokes three phases on the item P: *completer, predictor* and *scanner*. Because of the derivation of traces (ε) from the u-triples in T-stack, we need to add some code (the so-called pseudo-phases) that deals with completion, prediction and scanning of these entities.

Completer: When δ is an empty sequence (all the d-quad sequence has been traversed) and the top of T-stack is the empty set $\varnothing$ (all the triples concerning this item have been satisfied), the dotted rule has been completely analyzed. The completer looks for the items in S_j which were waiting for completion (*return items*; j is the return Position of the item P). The return items must contain a dotted rule where the dot immediately precedes a d-quad $\langle R,Y,u_x,\tau_x\rangle$, where Y is the head category of the dotted rule in the item P. Their generic form is $\langle x: X(\lambda \bullet \langle R,Y,u_x,\tau_x\rangle\ \zeta), j', \mu', \nu', \text{T-stack}'\rangle$. These items from S_j are inserted into S_i after having advanced the dot to the right of $\langle R,Y,u_x,\tau_x\rangle$. Before inserting the items, we need updating the T-stack component, because some u-triples could have been satisfied (and, then, deleted from the T-stack). The new T-stack" is the T-stack of the completed item after popping the top element $\varnothing$.

Predictor: If the dotted rule of P has a d-quad $\langle R_\delta,Z_\delta,u_\delta,\tau_\delta\rangle$ immediately after the dot, then the parser is expecting a subtree headed by a word of category Z_δ. This expectation is encoded by inserting a new item (*predicted item*) in the set, for each rule associated with Z_δ (of the form $z{:}Z_\delta(\theta)$). Again, each u-index of the new item (d-quad sequence θ) is replaced by a progressive integer. The ν-index component of the predicted item is set to u_δ. Finally, the parser prepares the new T-stack', by pushing the new u-triples introduced by τ_δ, that are to be satisfied by the items predicted after the current item. This operation is accomplished by the primitive PUSH-UNION, which also accounts for the non repetition of u-triples in T-stack. As stated in the derivation relation through the UNION operation, there cannot be two u-triples with the same relation and syntactic category in T-stack. In case of a repetition of a u-triple, PUSH deletes the old u-triple and inserts the new one (with the same u-index) in the topmost set. Finally, INSERT joins the new item to the set S_i.

The *pseudopredictor* accounts for the safisfaction of the the u-triples when the appropriate conditions hold. The current d-quad in P, $\langle R_\delta,Z_\delta,u_\delta,\tau_\delta\rangle$, can be the dependent which satisfies the u-triple $\langle u,R_\delta,Z_\delta\rangle$ in T-stack (the UNION operation gathers all the u-triples scattered through the T-stack): in addition to updating T-stack (PUSH($\varnothing$,T-stack)) and inserting the u-index u_δ in the ν component as usual, the parser also inserts the u-index u in the μ component to coindex the appropriate distant element. Then it inserts an item (*trace item*) with a fictitious dotted dependency rule for the trace.

Scanner: When the dot precedes the symbol #, the parser can scan the current input word w_i (if y, the head of the item P, is equal to it), or pseudoscan a trace item, respectively. The result is the insertion of a new item in the subsequent set (S_{i+1}) or the same set (S_i), respectively.

At the end of the external loop (*termination*), the sentence is accepted if an item of a root category Q with a dotted rule completely recognized, spanning the whole sentence (Position=0), an empty T-stack must be in the set S_n.

3.1. An example

In this section, we trace the parsing of the sentence "Beans I know John likes". In this example we neglect the problem of subject-verb agreement: it can be coded by inserting the AGR features in the category label (in a similar way to the +EX feature in the grammar G_1); the comments on the right help to follow the events; the separator symbol | helps to keep trace of the sets in the stack; finally, we have left in plain text the d-quad sequence of the dotted rules; the other components of the items appear in boldface.

S_0

<know: V+EX (• <VISITOR, N, 1, ∅> (initialization)
<SUBJ, N, 0, ∅>
#
<SCOMP, V, 0, <1, OBJ, N>>),
0, 0, 0,[]>

<likes: V (• <SUBJ, N, 0, ∅> (initialization)
#
<OBJ, V, 0, ∅>),
0, 0, 0,[]>

<beans: N (• #), 0, 0, 1,[∅]> (predictor "know")

<I: N (• #), 0, 0, 1, [∅]> (predictor "know")

<John: N (• #), 0, 0, 1, [∅]> (predictor "know")

<beans: N (• #), 0, 0, 0, [∅]> (predictor "likes")

<I: N (• #), 0, 0, 0, [∅]> (predictor "likes")

<John: N (• #), 0, 0, 0, [∅]> (predictor "likes")

S_1 [beans]

<beans: N (# •), 0, 0, 1, [∅]> (scanner)

<beans: N (# •), 0, 0, 0, [∅]> (scanner)
<know: V+EX (<VISITOR, N, 1, ∅> (completer "beans")
• <SUBJ, N, 0, ∅>
#
<SCOMP, V, 0, <1, OBJ, N>>),
0, 0, 0, []>
<likes: V (<SUBJ, N, 0, ∅> (completer "beans")
• #
<OBJ, V, 0, ∅>),
0, 0, 0, []>
<beans: N (• #), 1, 0, 0,[∅]> (predictor "know")
<I: N (• #), 1, 0, 0, [∅]> (predictor "know")
<John: N (• #), 1, 0, 0, [∅]> (predictor "know")

S_2 [I]

<I: N (# •), 1, 0, 0, [∅|∅]> (scanner)
<know: V+EX (<VISITOR, N, 1, ∅> (completer "I")
<SUBJ, N, 0, ∅>
• #
<SCOMP, V, 0, <1, OBJ, N>>),
0, 0, 0, []>

S_3 [know]

<know: V+EX (<VISITOR, N, 1, ∅> (scanner)
<SUBJ, N, 0, ∅>
#
• <SCOMP, V, 0, <1, OBJ, N>>),
0, 0, 0, []>
<likes: V (• <SUBJ, N, 0, ∅> (predictor "know")
#
<OBJ, V, 0, ∅>),
3, 0, 0, [{<1, OBJ, N>}]>
<beans: N (• #), 3, 0, 2,[{<1, OBJ, N>}|∅]> (p. "know")
<I: N (• #), 3, 0, 2, [{<1, OBJ, N>}|∅]> (p. "know")
<John: N (• #), 3, 0, 2, [{<1, OBJ, N>}|∅]> (p. "know")
<beans: N (• #), 3, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")
<I: N (• #), 3, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")
<John: N (• #), 3, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")

S_4 [John]

<John: N (# •), 3, 0, 2, [{<1, OBJ, N>}|∅]> (scanner)
<John: N (# •), 3, 0, 0, [{<1, OBJ, N>}|∅]> (scanner)
<know: V+EX (<VISITOR, N, 2, ∅> (completer "John")
• <SUBJ, N, 0, ∅>
#
<SCOMP, V, 0, <2, OBJ, N>>),
3, 0, 0, [{<1, OBJ, N>}]>
<likes: V (<SUBJ, N, 0, ∅> (completer "John")
• #
<OBJ, V, 0, ∅>),
3, 0, 0, [{<1, OBJ, N>}]>

S_5 [likes]

<likes: V (<SUBJ, N, 0, ∅> (scanner)
#
• <OBJ, N, 0, ∅>),
0, 0, 0, [{<1, OBJ, N>}]>
<beans: N (• #), 5, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")
<I: N (• #), 5, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")
<John: N (• #), 5, 0, 0, [{<1, OBJ, N>}|∅]> (p. "likes")
<ε: N (• #), 5, 0, 0, [∅]> (pseudopredictor)
<ε: N (# •), 5, 0, 0, [∅]> (pseudopredictor)
<likes: V (<SUBJ, N, 0, ∅> (completer)
#
<OBJ, N, 0, ∅>•),
3, 0, 0, [∅]>
<know: V+EX (<VISITOR, N, 1, ∅> (completer)
<SUBJ, N, 0, ∅>
#
<SCOMP, V, 0, <1, OBJ, N>> •),
0, 0, 0, []>

3.2. Complexity results

The parser has a polynomial complexity. The space complexity of the parser. i.e. the number of items, is $O(n^{3+|D||C|})$. Each item is a 5-tuple <Dotted-rule, Position, μ-index, ν-index, T-stack>: Dotted rules are in a number which is a constant of the grammar, but in off-line parsing this number is bounded by **O(n)**. Position is bounded by **O(n)**. μ-index and ν-index are two integers that keep trace of u-triple satisfaction, and do not add an own contribution to the complexity count. T-stack has a number of elements which depends on the maximum length of the chain of predictions. Since the number of rules is O(n), the size of the stack is O(n). The elements of T-stack contain all the u-triples introduced up to an item and which are to be satisfied (deleted) yet. A u-triple is of the form <u,R,Y>: u is an integer that is ininfluent, R∈D, Y∈C. Because of the PUSH-UNION operation on T-stack, the number of possible u-triples scattered throughout the elements of T-stack is |D||C|. The number of different stacks is given by the dispositions of |D||C| u-triples on O(n) elements; so, $\mathbf{O(n^{|D||C|})}$. Then, the number of items in a set of items is bounded by $O(n^{2+|D||C|})$ and there are **n** sets of items ($\mathbf{O(n^{3+|D||C|})}$).

The time complexity of the parser is $\mathbf{O(n^{7+3|D||C|})}$. Each of the three phases executes an INSERT of an item in a set. The cost of the INSERT operation depends on the implementation of the set data structure; we assume it to be linear ($O(n^{2+|D||C|})$) to make easy calculations. The phase *completer* executes at most $O(n^{2+|D||C|})$ actions per each pair of items (two for-loops). The pairs of items are $O(n^{6+2|D||C|})$. But to execute the action of the completer, one of the sets must have the index equal to one of the positions, so $O(n^{5+2|D||C|})$. Thus, the *completer* costs $\mathbf{O(n^{7+3|D||C|})}$. The

phase *predictor*, executes O(n) actions for each item to introduce the predictions ("for each rule" loop); then, the loop of the pseudopredictor is O(|D||C|) (UNION+DELETE), a grammar factor. Finally it inserts the new item in the set ($O(n^{2+|D||C|})$). The total number of items is $O(n^{3+|D||C|})$ and, so, the cost of the predictor $\mathbf{O(n^{6+2|D||C|})}$. The phase *scanner* executes the INSERT operation per item, and the items are at most $O(n^{3+|D||C|})$. Thus, the scanner costs $\mathbf{O(n^{5+2|D||C|})}$. The total complexity of the algorithm is $O(n^{7+3|D||C|})$.

We are conscious that the (grammar dependent) exponent can be very high, but the treatment of the set data structure for the u-triples requires expensive operations (cf. a stack). Actually this formalism is able to deal a high degree of free word order (for a comparable result, see (Becker, Rambow 1995)). Also, the complexity factor due to the cardinalities of the sets D and C is greatly reduced if we consider that linguistic constraints restrict the displacement of several categories and relations. A better estimation of complexity can only be done when we consider empirically the impact of the linguistic constraints in writing a wide coverage grammar.

4. Conclusions

The paper has described a dependency formalism and an Earley-type parser with a polynomial complexity.

The introduction of non lexical categories in a dependency formalism allows the treatment of long distance dependencies and of free word order, and to aovid the NP-completeness. The grammar factor at the exponent can be reduced if we furtherly restrict the long distance dependencies through the introduction of a more restrictive data structure than the set, as it happens in some constrained phrase structure formalisms (Vijay-Schanker, Weir 1994).

A compilation step in the parser can produce parse tables that account for left-corner information (this optimization of the Earley algorithm has already been proven fruitful in (Lombardo, Lesmo 1996)).

References

Becker T., Rambow O., *Parsing non-immediate dominance relations*, Proc. IWPT 95, Prague, 1995, 26-33.

Covington M. A., *Parsing Discontinuous Constituents in Dependency Grammar*, **Computational Linguistics 16**, 1990, 234-236.

Earley J., *An Efficient Context-free Parsing Algorithm*. **CACM 13**, 1970, 94-102.

Eisner J., *Three New Probabilistic Models for Dependency Parsing: An Exploration*, Proc. COLING 96, Copenhagen, 1996, 340-345.

Fraser N.M., Hudson R. A., *Inheritance in Word Grammar*, **Computational Linguistics 18**, 1992, 133-158.

Gaifman H., *Dependency Systems and Phrase Structure Systems*, **Information and Control 8**, 1965, 304-337.

Hahn U., Schacht S., Broker N., *Concurrent, Object-Oriented Natural Language Parsing: The ParseTalk Model*, CLIF Report 9/94, Albert-Ludwigs-Univ., Freiburg, Germany (also in **Journal of Human-Computer Studies**).

Hudson R., *English Word Grammar*, Basil Blackwell, Oxford, 1990.

Kwon H., Yoon A., *Unification-Based Dependency Parsing of Governor-Final Languages*, Proc. IWPT 91, Cancun, 1991, 182-192.

Lombardo V., Lesmo L., *An Earley-type recognizer for dependency grammar*, Proc. COLING 96, Copenhagen, 1996, 723-728.

Mel'cuk I., *Dependency Syntax: Theory and Practice*, SUNY Press, Albany, 1988.

Milward D., *Dynamic Dependency Grammar*, **Linguistics and Phylosophy**, December 1994.

Nasr A., *A formalism and a parser for lexicalized dependency grammar*, Proc. IWPT 95, Prague, 1995, 186-195.

Neuhaus P., Broker N., *The Complexity of Recognition of Linguistically Adequate Dependency Grammars*, Proc. ACL/EACL97, Madrid, 1997, 337-343.

Neuhaus P., Hahn U., *Restricted Parallelism in Object-Oriented Parsing*, Proc. COLING 96, Copenhagen, 1996, 502-507.

Rambow O., Joshi A., *A Formal Look at Dependency Grammars and Phrase-Structure Grammars, with Special Consideration of Word-Order Phenomena*, Int. Workshop on The Meaning-Text Theory, Darmstadt, 1992.

Schabes Y., *Mathematical and Computational Aspects Of Lexicalized Grammars*, Ph.D. Dissertation MS-CIS-90-48, Dept. of Computer and Information Science, University of Pennsylvania, Philadelphia (PA), August 1990.

Sgall P., Haijcova E., Panevova J., *The Meaning of Sentence in its Semantic and Pragmatic Aspects*, Dordrecht Reidel Publ. Co., Dordrecht, 1986.

Sleator D. D., Temperley D., *Parsing English with a Link Grammar*, Proc. of IWPT93, 1993, 277-291.

Vijay-Schanker K., Weir D. J., *Parsing some constrained grammar formalisms*, **Computational Linguistics 19/4**, 1994