Parsing with Dependency Relations and Robust Parsing

Jacques Courtin, Damien Genthial
CLIPS - IMAG CAMPUS
BP 53
38040 GRENOBLE CEDEX 9
Phone: +33 476 51 49 15
E-Mail: Jacques.Courtin@imag.fr, Damien.Genthial@imag.fr

Abstract

After a short recall of our view of dependency grammars, we present two dependency parsers. The first uses dependency relations to have a more concise expression of dependency rules and to get efficiency in parsing. The second uses typed feature structures to add some semantic knowledge on dependency trees and parses in a more robust left to right manner.

1. Introduction

Our team has been working with dependency grammars for more than twenty-five years (Courtin 73). Two dependency parsers built by our team are presented in this paper. The first one uses the notion of dependency relations in order to implement dependency grammars efficiently; it is described in the first part of the text. The second one was built with the following objectives: adding the use of some semantic knowledge in the process of syntactic parsing and obtaining a robust parser (second part of the text).

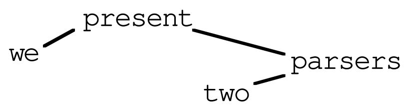
2. Parsing with dependency relations

The linguistic model we use for dependency is inspired by the Tesnière model (Tesnière 59), which we will recall shortly in order to define precisely our terminology.

2.1. The linguistic model

Relationship between words is the fundamental concept associated with dependency structures (DS). Given two words of the language, a relation is established between them, defining a dominated word (or dependent) and a dominating word (or governor). This relation can be represented by an arc between two nodes, where each node is labelled by a word. The arc descends from the governor to the dependent.

Example: the dependency structure for the sentence « we present two parsers »:



We can also use a linear notation with brackets and write: `(we) present ((two) parsers)`. But the graphical representation is more readable and shows clearly the hierarchy between the governor and its dependents, which of course, can also have dependents.

Dependency grammars

A dependency grammar (formalism used by (Hays 64)) on a vocabulary V is made of:

- a family of parts C_i of V such that the union of C_i is equal to V.
- a set of rules, each having one of the two following forms:

  i) $*$ (X)

  ii) X (X_1 ... X_i $*$ X_{i+1} ... X_n)

C_i are *word classes* or *lexico-syntactic categories* and are denoted by their name (Determiner, Noun, Adjective,...). X_i in the rules above are category names.

The star shows the place of the governor relatively to its dependents, so in a type ii) rule, $X_1 ... X_i$ are *left dependents* of the X governor and $X_{i+1} … X_n$ are its *right dependents*.

When n = 0, the rule is written X(*) and is a *terminating* rule; type i) rules are *initial* rules.

Grammar example:
We use the following categories: Determiner (`D`), Noun (`N`), Adjective (`A`), Verb (`V`).

```
*(V)
V(N,*,N)
N(D,*)     N(D,A,*)  N(A,*)
D(*)       V(*)      A(*)      N(*)
```

```
V={drinks, eats}      D={the, a}
N={dog, cat, cup, milk}
A={black, white, hot}
```

With this grammar one can build the structure:

```
               drinks
              /      \
           cat        milk
          /   \          \
     the   black          hot
```

Generation

Dependency grammars are generative, working with the following generating rules:

a) choose a type i) rule (which determines the main governor),

b) choose and apply type ii) rules until we obtain a complete structure, entirely made of terminating rules.

With the example grammar above, we can make the following derivation (which matches the sentence: « the black cat drinks hot milk »):

```
*(V)
*(V(N,*,N))
*(V(N(D,A,*),*,N))
*(V(N(D,A,*),*,N(A,*)))
*(V(*)(N(D,A,*),*,N(A,*)))
......
*(V(*)(N(*)(D(*),A(*),*),
        *,
        N(*)(A(*),*)))
```

Remark:
For a given governor, the dependency grammar must contain as many rules as there are possible configurations of dependents below this governor. For example, if we want nominal phrases with at least a noun, an optional determiner and 0,1 or 2 adjectives before the noun, we will have the grammar:

```
N(*)       N(A,*)     N(A,A,*)
N(D,*)     N(D,A,*)   N(D,A,A,*)
```

The formalism proposed below shows a better way to describe the same things.

2.2. Dependency relations

The method used in the PILAF[1] system (Courtin 77) to build dependency structures is a direct analysis: we transform the input word chain in a dependency tree by using a form of dependency grammar and no intermediate structure. But the algorithm does not directly use Tesnière type dependency grammars because, as we seen before, these grammars impose a combinatorial description of all the possible configurations of dependents for a given governor. To overcome this drawback, we introduce *dependency relations* between two lexico-syntactic categories.

Example:
To say that N governs the A we simply write N -> A.

Dependency Relations (DR) must not only code the relation itself but also:

- the relative positions of the dependent and the governor: is it a left dependent or a right one ?
- the relative positions of all dependents of a given governor.

Example:
We want to describe the sentence « The black cat drinks hot milk » which gives the sequence of categories:

```
D A N V A N
```

and the dependency tree given above.

Dependency relations must stipulate that a noun can appear on the left or on the right below a verb and that below a noun, the determiner precedes the adjective. So we attach to each relation an vector of integers (either positive or negative) and we write:

GOUV -> DEP := (x_1 ,...,x_n),

which says that we can have 0,1…n dependents of the DEP category below the governor of the GOV category.

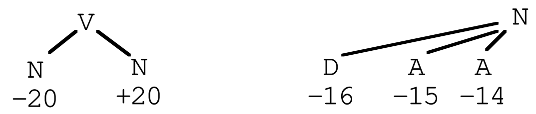
The integers are presented in ascending order, showing the relative position of DEP below GOUV. For any given governor, the integer values also determine the relative positions of all its different possible dependents.

Example:

```
N -> A := (-14, -15)
N -> D := (-16)
V -> N := (-20, + 20)
```

Positive integers concern right dependents and negative integers left dependents. The integer of the second relation stipulate that the determiner, if any, will be placed before the adjectives, because -16 is less than -15 and -14. From the first relation we can see that no word can be placed, below the noun, between the two adjectives (there is no integer between - 15 and -14). These relations can be drawn as the following trees:

[1]Procédures Interactives Linguistiques Appliquées au Français (Interactive Linguistic Procedures Applied to French)



An important thing to be noted is that each integer position gives the possibility for a dependent to be present at that position, but never imposes that presence.

So the three relations above are equivalent to the following dependency grammar:

```
N(A,*)      N(A,A,*)
N(D,*)      N(D,A,*)  N(D,A,A,*)
V(N,*)      V(*,N)    V(N,*,N)
N(*)        A(*)      D(*)        V(*)
```

It can be noted that these relations are in some sense similar to disjunctive forms of Sleator's link grammars (Sleator and Temperley 91).

2.3. Parsing algorithm

This algorithm supposes that the morphological step is finished and that it has produced the sequence of lexico-syntactic categories for the input sentence, each word corresponding to one category – or several if the word is ambiguous.

So the parser's inputs are:

- the sequence $X_1 ... X_n$ of categories computed by the morphological parser;
- the set of dependency relations and the associated integer vectors.

We add to the X_i sequence the pseudo category SENT=X_0 which will help in determining the possible governor of the sentence (to initiate the parsing process). If, for example, possible main governors of a sentence are coordination conjunction (C) and verb, we will have the relations:

```
SENT -> V := (+1)
SENT -> C := (+1)
```

As we can have only one governor for a sentence, these two relations are mutually exclusive. This is expressed by the value of the integer: +1, which is the same for the two possible dependents of SENT.

In order to build the dependency tree (or trees) associated with the given sequence of categories, the parser first initializes the square array of figure 1.

As (Sleator And Temperley 91), we only want *projective* structures (or *planar* structures), i.e. trees which can be traversed by a left to right infix algorithm to find the original linear order of the sentence. The motivations for this limitation to projective structures are the following:

- it is important to be able to retrieve, from the tree, the original linear form of the sentence;
- this limitation leads to greater parsing efficiency: for each governor, the search for its dependents will be made in two separate spaces: a left and a right space.

GOV DEP	X_0 SENT	X_1	X_2		X_N
X_0 SENT	∅	P_{01}	P_{02}		P_{0n}
X_1	P_{10}	∅	P_{12}		P_{1n}
X_2	P_{20}	P_{21}	∅		P_{2n}
X_N	P_{n0}	P_{n1}	P_{n2}		∅

P_{ij}: set of integers determined by the relations X_j -> X_i.

P_{ii}: ∅

Figure 1 : Square array for a sentence

So for a given governor X_j, all its left dependents must have an index $i < j$ (the index order matches the word order in the sentence). The same is true for right dependents, with index $k > j$. So we can remove from the top-right triangle of the array all positive numbers and from the bottom-left triangle all negative ones. We then have the two properties:

- $1 \leq i, j \leq n$, $i > j$, if $P_{ij} \neq \emptyset$ then $\forall\, p \in P_{ij}$, we have $p > 0$.
- $1 \leq i, j \leq n$, $i < j$, if $P_{ij} \neq \emptyset$ then $\forall\, p \in P_{ij}$, we have $p < 0$.

After having initialized the array and removed useless parts of it, the parser builds, with a descendant and recursive algorithm, all dependency structures compatible with the array:

a) For each possible governor of the sentence (SENT column):

  b) build all left sub-trees and all right sub-trees (recursively);

  c) build the final structures by merging the partial right and left ones.

One can say that we catch the SENT category and « pull » the structures out of the array. The algorithm succeeds if at least one « pulled » structure contains all the words of the input sentence.

With real sentences, of course, we have lexical ambiguities or structural ambiguities. In both cases, the algorithm is non-deterministic and builds all possible solutions by blind combinatorial enumeration.

Dependency relations, associated with the algorithm described above, constitute a grammatical model with very few constraints. We can quickly state that the parser will succeed on more sentences than the language sentences. This feature can be viewed as an advantage in the framework of a man-machine communication system, where the essential quality of an utterance is to be interpretable, even if it is not syntactically correct: « Close file », for example, is ungrammatical but we can interpret it and execute the associated command.

On the contrary, this lack of constraints is penalizing efficiency: the algorithm will build a lot of incorrect structures because we can not state for example, that a given governor must have at least one dependent at that position, that a given relation only apply in a given context,...

These limits and the necessary addition of some semantic knowledge in the syntactic parsing process lead us to design the new method for dependency tree construction presented in the second part.

2.4. Conclusion

Despite its relatively limited power of expression, this parser builds dependency structures extremely quickly (« instantaneously » on a personal computer) as long as the input sentence is not too long and not too ambiguous (say when the number of produced trees is less than 20).

The parser has been put to use in a system for detection and correction of syntactic errors (Strube de Lima 90). The main purpose was to check the numerous concordancy rules for gender, number and person in written French sentences. For this type of application, it was of course essential for the parser not to take into account morphological properties of words while building dependency structures.

By its lack of constraints and its high practical efficiency, this algorithm could be used in applications for man-machine interfaces where exchanges are short and language often approximative.

3. Robust Parsing

The use of the preceding parser in a system for detection and correction of syntactic errors in French has raised the following problems:

- even for a simple task such as detection and correction of agreement errors in written texts, you need a powerful parsing mechanism able to determine, for example, the antecedent of a relative pronoun;
- a system for error correction can not rely on the correctness of the inputs in order to build a structure which is essential to make a minimal work. So you have to improve the knowledge of the system, i.e. in our case, to add some semantic information on words in order to determine more precisely the relations between them;
- the syntactic parser of a such system must also be robust and produce an output even if the input is completely ill-formed.

These problems lead us to define a new dependency parser which will be able to manipulate some semantic information and which will be error resistant. This work results in a prototype called CADeT[2] of a dependency tree transducer, which we will describe in the following sections.

3.1. A language for writing dependency grammars

We have attempted to design a language for the description of dependency structures retaining the precision of Tesnière's grammars, but more appropriate for automatic treatment. Our basic idea is that the governor-dependent relation should not be expressed for two categories in general, but for two words which are instanciations of these categories in a given sentence. We therefore think it is necessary, when describing a governor-dependent relation, to indicate the context in which the relation is valid.

To build dependency structures, we must be able to determine, for any two words, caracterized by their lexical category: determiner, noun, verb, …, which one governs the other. More generally, given two dependency trees, we must know how to merge them into a unique tree.

Example:

```
  N - V   ──►    V
 /     \        / \
D       N      N   N
              /
             D
```

`(D)N, V(N) -> ((D)N)V(N)`

We have defined a language based on rewriting rules; each rule applies to a dependency **forest**

---

[2] Constructeur d'Arbres de DEpendances Typés (Typed Dependency Trees Builder).

and produces a dependency **tree.** A set of such rules constitutes a dependency grammar which can be applied to a sentence by means of an interpreter. This interpreter is in fact a tree-transducer driven by the rules.

Example of a simple rule: (the "--" begins comments)

```
N_V [ -- Name of the rule
(1:{N}, (0, $F:{pv})2:{V}) -- Forest
=>
( ( 1, $F ) 2 ) ] -- Resulting tree
```

This rule applies to any forest which includes a sequence of an `N` and a `V`, whose left dependents are only preverbal particles `pv`. It builds a new tree where the `N` is added as a dependent of the `V`.

The advantage of these rules, compared to simple binary relations, is that it is possible to express the context of each category which appears. It is thus possible to restrict a governor to one or two dependents only, or to forbid more than one occurrence of a given category,... One can also define linked pairs of binary relations, as for coordination conjunctions (`C`):

```
N_C [
(1:{N}, 2:{C}, 3:{N})
=>
( ( 1 ) 2 ( 3 )) ]
```

On the other hand, they present the drawback of the primitive dependency grammars: there must be a rule for almost every pair of lexical categories (LC). To avoid this problem, we have chosen to use a **hierarchy** of LCs instead of the usual **linear set** of LCs (Genthial & al. 90). This hierarchy is a set, partially ordered by the `is-a` relation (figure 2).

We can, in this manner, express very general rules like the two given above (`N_V` and `N_C`) or more specific ones like:

```
aux_ppas [
(1:{xbe; xhave}, 2:{pastp})
=>
( ( 1 ) 2 ) ]
```

By means of `is-a({cnoun, pnoun}, N)` and `is-a({xbe, xhave, verb, pastp}, V)` relations, the `N_V` rule for instance may be applied to all the following pairs of categories:

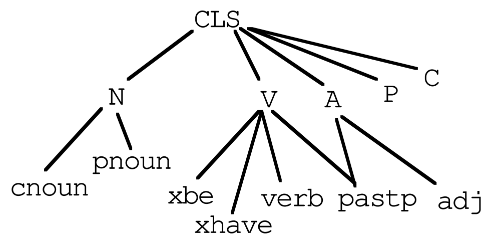
```
(cnoun, xbe)        (pnoun, xbe)
(cnoun, xhave)      (pnoun, xhave)
(cnoun, verb)       (pnoun, verb)
(cnoun, pastp)      (pnoun, pastp)
```

We can thus define a set of basic categories which describe words in a very specific way, and use these categories for lexical indexing. The categories can then be grouped in « meta-categories » according to the structures we want to build. Finally, we can write the rules which effectively build these structures.

We can also write grammars in an incremental fashion, starting with the highest categories (e.g. N, V, A, C, P) then testing the rules on our corpus, progressively adding more precise rules for the lowest categories to treat specific phenomena.

So, by using this method, we can avoid the usual compromise between a very fine set of LCs (which multiplies morphological ambiguities and syntactic rules) and a very general set (which multiplies syntactic ambiguities). We also obtain a fairly robust syntactic parsing: all unknown words are given the most general category (`CLS`), to which any rule can apply, thus an unknown word does not stop the parsing process.

Similar type hierarchies have already been used in work on language semantics to represent the taxonomy of semantic types. We shall therefore use the same formalism for the representation of syntactic and semantic knowledge (see §3.3).



We use the following abbreviations: cnoun and pnoun for common and proper nouns, xbe and xhave for the auxiliaries be and have, pastp for past participle, adj for adjective, P for preposition and C for coordination conjunction.

Figure 2: Example of hierarchy

3.2. Building dependency structures

Given a set of rewriting rules, the tree transducer proceeds by a left to right scanning of the input text. Each time a word is recognized by the morphological parser, it is transmitted to the syntactic module which includes it in the current state of the analysis. As the data manipulated by the tree transducer must be trees or forests, each word is transformed in a one node tree, where the root bears the information associated to the word.

In order to manage multiple interpretations of the same word or of the same sentence, the transducer maintains a list of forests where each forest is a possible interpretation of the sentence. These forests, which are the current state of the analysis, are called *stacks* because each time a new word is recognized, a one node tree is pushed on each forest and the parsing always resumes on the top of each forest.

Given a list of stacks, the transducer applies each applicable rule to the top of all stacks and each time a rule applies, a new stack is produced and added at the end of the list. Doing so, the transducer will also apply the rules to the new stacks produced, cyclically. If more than one rule applies to a particular stack, more than one stack will be produced, but if at least one rule applies to a stack, this stack will be removed from the list.

Example: (adapted from French)
We consider only four categories: `D`, `A`, `N`, `V` (for determiner, adjective, noun and verb) and we give the following very simple rules:

```
D_N [(1:{D}, 2:{N}) => ((1) 2) ]
A_N [(1:{A}, 2:{N}) => ((1) 2) ]
N_A [(1:{N}, 2:{A}) => (1 (2)) ]
N_V [(1:{N}, 2:{V}) => ((1) 2) ]
V_N [(1:{V}, 2:{N}) => (1 (2)) ]
```

Figure 3 shows the evolution of the list of stacks during the parsing of the French nominal phrase: « la belle ferme » which is ambiguous and leads to the following sequence of categories:

$$D \begin{Bmatrix} A \\ N \end{Bmatrix} \begin{Bmatrix} A \\ N \\ V \end{Bmatrix}$$

We first introduce the word « la » as a one node tree bearing the `D` category. As no rule can apply to this tree, we then introduce the word « belle » which is ambiguous. The ambiguity gives two forests which are described on list (1). The `D_N` rule applies to this list and gives list (2).

Introducing the word « ferme » leads to list (3), on which we detail rule application. So the rule `A_N` applied to the second stack of the list produces a new forest (or stack) which is appended to the list. When the transducer ends with the original list, it finds the new produced stacks and proceeds with them, applying grammar rules. The `D_N` rule will then be applied to the new produced forest `(D, (A)N)`. The process stops when the transducer reaches the end of the list and, after removing the stacks where a rule has applied, we obtain list (4).

A correct interpretation (according to a given grammar) of the input sentence can be found in each stack which contains exactly one tree: this tree is a dependency structure of the sentence.

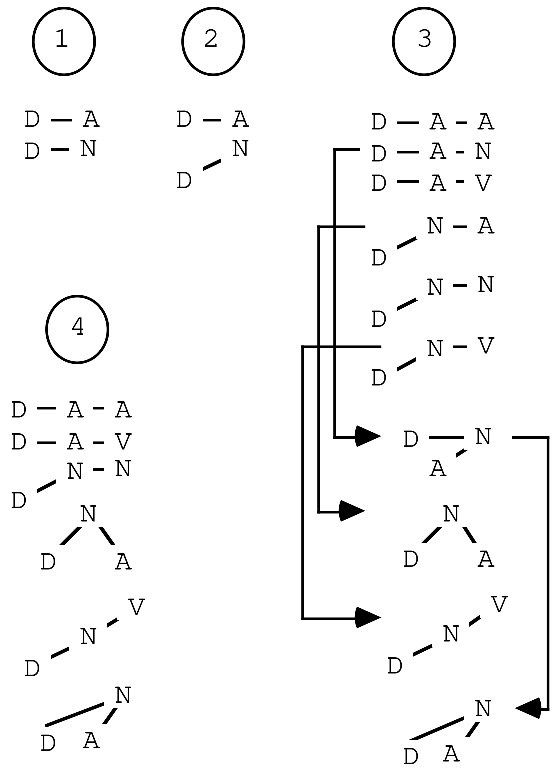


Figure 3: Stacks evolution

Our example gives three correct structures:

```
(la)belle(ferme)  the firm beauty
((la)belle)ferme  the beauty closes
(la,belle)ferme   the beautiful farm
```

The algorithm is guaranted to stop because we have added a constraint: rewriting rules are written in such a way that the length of a stack must reduce each time a rule is applied to it. A detailed discussion of termination and an evaluation of the algorithm can be found in (Genthial 91).

3.3. Type hierarchies

We have chosen to represent knowledge about words and trees with a unique formalism: *Ψ-terms* (Aït-Kaci 84). Ψ-terms are typed features structures which permit the description of types (in the sense of classical programming languages such as Pascal), i.e. sets of values.

Example:

```
UL(lex => "eats";
    cat => verb;
    subj => UL(sem => S:ANIMATE);
    obj => UL(sem => O:EATABLE);
    sem => INGEST(agent => S;
                  patient => O))
```

The use of reference tags like S or O allows structure sharing, so Ψ-terms are not trees but graphs.

Simple types are defined in the *signature* which is a set partially ordered by the is-a relation. This order is extended to Ψ-terms by the unique operation used to manipulate them: *unification*. The unification of two simple types is defined as the set of lower bounds of these two types (in the is-a relation). Unification allows implicit inheritance of properties, and can be efficiently implemented (Aït-Kaci & al. 89).

In our parser, a Ψ-term is attached to each node of a tree and to transduction rules we have added expressions which enable us to test and modify those Ψ-terms. We can thus simultaneously build a syntactic structure (dependency tree) and a semantic structure (Ψ-term, which also contains morphological and syntactical information), and which is built by unification (see also (Hellwig 86) on the use of unification for dependency parsing).

Example of rules and application:
We have two words:

```
UL(lex => "dog";
   cat => cnoun;
   sem => CANINE)
UL(lex => "eats";
   cat => verb;
   subj => UL(sem => S:ANIMATE);
   obj => UL(sem => O:EATABLE);
   sem => INGEST(agent => S;
                 patient => O))
```

and the rule:

```
subject [ (1: {N}, 2:{V})
/Unif(1, 2.subj)/ -- Conditions
=>
( ( 1 ) 2 );
ASSIGN(2.subj, 1); ]    -- Actions
```

The root of the resulting tree is decorated by:

```
UL(lex => "eats";
   cat => verb;
   subj => UL(lex => "dog";
              cat => cnoun;
              sem => S:CANINE);
   obj => UL(sem => O:EATABLE);
   sem => INGEST(agent => S;
                 patient => O))
```

3.4. Conclusion

The use of a category hierarchy simplifies the writing of the rules and introduces a way of manipulating unknown words which is not part of the mechanisms of the system but which is integrated in the objects it manipulates. We can then write rules without thinking about ill-formedness (i.e. it is not necessary to make the rules tolerant because the tolerance is implicit in the system).

More generally, the use of unification in conjunction with dependency parsing allow to build syntactic structures efficiently while having the possibility to make very fine descriptions with Ψ-terms.

References

Hassan Aït-Kaci (1984). *A Lattice-Theoretic Approach to Computation Based on a Calculus of Partially-Ordered Type Structures.* Ph. D., University of Pennsylvania 1984.

Hassan Aït Kaci et al. (1989). *Efficient implementation of Lattice Operations.* ACM Transactions on Programming Languages and Systems **11**:1, pp. 116-146.

Jacques Courtin (1973). *Un analyseur syntaxique interactif pour la communication homme-machine.* Intl Conference of Computational Linguistics, Pise, Italy, August 1973, Vol. 1.

Jacques Courtin (1977). *Algorithmes pour le traitement interactif des langues naturelles*. Thèse d'état, Grenoble I, Octobre 1977.

Damien Genthial, Jacques Courtin et Irène Kowarski (1990). *Contribution of a Category Hierarchy to the Robustness of Syntactic Parsing.* 13th CoLing, Helsinki, Finland, August 1990, Vol. 2, pp 139-144.

Damien Genthial (1991). *Contribution à la construction d'un système robuste d'analyse du français.* Thèse, Université Joseph Fourier, 10 janvier 1991.

D. Hays (1964). *Dependency theory : a formalism and some observations.* Language **40**, pp. 511-525.

Peter Hellwig (1986). *Dependency Unification Grammar.* 11th CoLing, Bonn, FRG, August 1986, pp 195-198.

Daniel Sleator et Davy Temperley (1991). *Parsing English with a Link Grammar.* Technical Report CMU-CS-91-196, School of Computer Science, Pittsburgh, October 1991.

Véra Lucia Strube de Lima (1990). *Contribution à l'étude du traitement des erreurs au niveau lexico-syntaxique dans un texte écrit en français.* Thèse, Université Joseph Fourier, Mars 1990.

Lucien Tesnière (1959). *Eléments de syntaxe structurale.* Klincksiek, Paris