

Constraint-Based Dependency Parsing

Sebastian Blohm

Universität des Saarlandes, 66041 Saarbrücken, Germany
sebastian@blomega.de

Abstract. Constraint-based dependency parsing is an approach to using constraint programming for natural language processing. The task of parsing an input sentence can be described as graph configuration problems in which grammatical knowledge plays the role of constraints on the structure of the graphs. In this paper it is shown how important requirements on parsing results can be modelled as finite set constraints and how the interaction of several configuration problems allows different kinds of grammatical information to contribute to the parse. An outlook is given on the XDG grammar formalism which incorporates a constraint-based description of linguistic knowledge.

1 Introduction

A grammar defines a language by identifying valid compositions of words to phrases and sentences. Parsing is the process of determining for an input expression if this expression is licensed by a given grammar. If this is the case, the structure that is imposed on the expression by the grammar can be output. This structure can be described using a *parse tree*. The task of parsing will be identified with the task of finding appropriate tree representations of the input.

Parsing is a subtask of many applications because determining such a structure is frequently crucial for further processing. Standard approaches to parsing involve incremental construction of parse trees starting at one end of the expression (bottom-up) or generating many parse trees and testing if they match the input expression (top-down).

Constraint-based dependency parsing constitutes an alternative approach in which the grammar defines a set of equations that need to be satisfied by any parse tree. Parsing an expression then consists of finding one or more parse trees that satisfy all these equations while reflecting the expression given. The techniques presented here are in application in the *Extensible Dependency Grammar (XDG)* which is designed for describing natural language.

1.1 Parsing Natural Language

Particularly challenging parsing tasks occur when natural language is processed. As the meaning of a sentence depends on its grammatical structure, a valid parse tree is the prerequisite for most higher level processing stages. However, as opposed to most artificial languages, natural language has not been designed

to be unambiguous and concise in any case. A natural language parser has to be able to cope with ambiguity in an efficient way. In particular, backtracking overhead has to be minimized.

Dependency Grammars. Dependency grammars form a particular type of grammar formalisms. Instead of identifying phrases and their constituents like for example in context-free grammars, they describe the relations between words. Each phrase has one word as a *head* and possibly a set of dependents, which are in turn heads of phrases. A head is said to (immediately) *dominate* the dependants. The dependency parse of a sentence can be described as a labelled tree with its words as nodes and edges from each head to its dependants. The syntactic relations among heads and dependents are given as edge labels.

Labelled Graphs. The parse trees used here are defined by a set of vertices V , a set of labels and a set of labelled directed edges $E \subseteq V \times L \times V$.

Immediate Dominance (ID) Trees. Immediate dominance trees play an important role in dependency parsing as they reflect the phrase structure of the input. They form a tree shaped instance of labelled graphs with edges directed away from the root. The set of nodes V of the parse trees used here consists of the words of the input expression. Edges establish *dominance* between phrases. Given a node $v \in V$ the phrase that is headed by v consists of v and all words that occur below v . The labels of the edges describe syntactical roles. In figure 1 the verb “sees” dominates its subject “Mary” and the phrase “the man with the telescope” which plays the syntactic role of an object. As far as it is considered here one will find deriving an immediate dominance tree a central task of parsing.

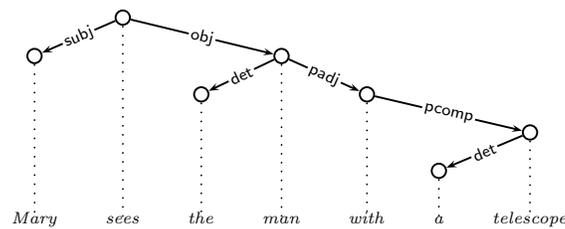


Fig. 1. One possible ID tree of “Mary sees the man with a telescope.”

Words have various grammatical properties that determine where they can appear in an immediate dominance tree. One of them is called their *valency*. The valency determines, which syntactic roles need to be filled. For example, each finite verb needs a subject. Hence, the valency of a finite verb will require the verb’s node in the immediate dominance tree to have an outgoing edge labelled *subj*. The properties of a word are determined by its *lexical entry*. Note that there

might be several possible lexical entries for each word in an input sentence. For example, from the word “walk” itself it is unclear if it occurs as a finite or infinite verb or even a noun. It is one task of the parser to identify the lexical entries of the input words to make sure that the parse tree derived does not conflict with the word’s properties.

Lexical Ambiguity. A phrase is lexically ambiguous if for one or more of its words there are more than one possible lexical entries that fit its position in the parse tree.

Structural Ambiguity. The second form of ambiguity results in parse trees of different shape. Structural ambiguity occurs whenever the grammar allows two different dominance structures of a given phrase. Figure 2 constitutes an alternative ID tree of the given sentence. The phrase “with a telescope” can either be dominated by “man” or by “sees”. Each parse tree corresponds to a different interpretation differing with respect to who is in hold of the telescope.

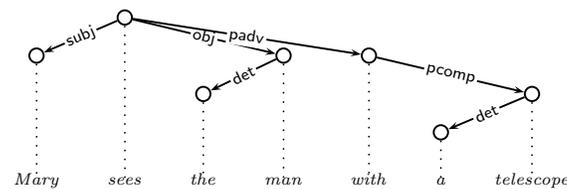


Fig. 2. Alternative ID tree of “Mary sees the man with a telescope.”

1.2 Constraint Programming

Constraint programming [1] is a programming paradigm developed for efficiently solving *constraint satisfaction problems (CSPs)*. The goal of solving a CSP is to find values for the problem’s variables that satisfy all given constraints. A constraint programming system allows the programmer to directly state the constraints. A module called the constraint solver then takes care of finding one or more solutions. To this end, the constraint solver syntactically transforms one or more constraints in order to derive stronger (more constraining) constraints (*propagation*) and tries how hypothetically restricting the set of possible variable assignments affects the set of solutions (*distribution*). The domain of the variables in question is in the simplest case a finite set of integers. Other domains are a set of sets of integers or a subset of the real numbers. Constraints can in the simplest case be expressed using equations ($X = 5$) or inequalities ($S \subseteq \{1, 2, 3\}$). Modern constraint solvers however allow constraints with more complex semantics.

When applied to parsing, the variables of a CSP belong to descriptions of the

parse trees and the constraints are given by the grammar as well as by the input expression. It is therefore of central concern when talking about constraint-based parsing, which constraints are needed to build a grammar that is appropriate for the case at hand.

1.3 About this paper

This seminar contribution is based on two papers [2],[3] that describe modelling for constraint-based dependency parsing. Section 2 presents a way of modelling parsing problems as constraints satisfaction problems. Details are given on several types of constraints that form the central part of this modelling. Section 3 illustrates how these constraints interact when parsing an example sentence. Section 4 introduces some aspects of the XDG grammar development framework before section 5 concludes the contribution.

2 Constraints for Parsing

Given the definition of a labelled graph it is easy to imagine a program to generate all finite labelled graphs. Obviously, the desired parse tree is among those graphs. It is now the goal to formulate constraints that only admit proper parse trees. Constraints that are concerned with the general well-formedness of the resulting parse trees are also called *principles*.

2.1 Requirements for Parsing Constraints

In this contribution, the focus is on four tasks that the principles need to fulfill:

1. *Exclude non-trees.* The graph describing the parsing results has to be a tree. To describe a valid dependency structure, each node has to have at most one parent, i.e. each word is only immediately dominated by one other word. Also, no cycles are allowed as a sub-phrase has to be strictly smaller than the phrase it is dominated by. Finally, there has to be exactly one root which is the node dominating the entire input.
2. *Exclude trees with improper lexical choice.* Each word has to be assigned a lexical entry. Obviously, the lexical entry has to be applicable for the input word. If the input word is “walk” the choice has to be restricted to the set of lexical entries for “walk”. From there on, it is required, that the values of all properties like for example valency will match the same lexical entry of this set. This way it is made sure that, if in the resulting parse tree the word “walk” plays a syntactic role that only finite verbs can play, it cannot have the valency of the infinite interpretation of “walk” at the same time.
3. *Enforce valency.* The edges of a parse tree have to reflect dependency as it is intended by the grammar. If, as in figure 2 the verb “sees” requires a subject and an object, the parse tree has to have the respective edges (e.g. subj and

- obj). At the same time non-sense edges have to be disallowed. For example “Mary” can not have an incoming edge labelled **det** for determiner. Trees disrespecting valency do not need to be considered as candidate parse trees.
4. *Enforce word order.* Word order is an important aspect during parsing. The choice of the syntactic role of a word is limited by its relative position in the input. In figure 2, “Mary” has to be the subject of “sees” because it occurs before the verb. Parse trees disrespecting aspects of word order have to be eliminated. Formulating the respective constraints turns out to be a non-trivial task in languages like German that allow relatively free word order or when parsing syntactic phenomena featuring non-coherent phrases (e.g. raising).

2.2 Tree Configuration Problems

In order to state the constraints, a more formal description of the problem is required. As it is the goal to compose a labelled tree that reflects the parsing result, tree configuration problems are defined as follows:

Tree Configuration Problem. Given a finite set of nodes V and a finite set of labels L a tree configuration problem is to find for all $l \in L$ the function $l(w) : V \rightarrow 2^V$ that assigns each node w the set of nodes to which there is an edge labelled l from w . For example $\text{subj}(\text{sees}) = \{\text{Mary}\}$ in figure 2.

2.3 Treeness Constraints

In order for a graph to be a tree, the following three conditions have to hold:

1. Each node has at most one incoming edge.
2. There is precisely one root (node without incoming edge).
3. There are no cycles.

To make it easier to define the treeness constraints formally the following abbreviating functions are defined: **daughters**, **down**, **eqdown** all of type $V \rightarrow 2^V$. They constitute the abstraction over all l in $l(w)$ as well the transitive and the reflexive-transitive closure of that abstraction. Therefore given a labelled graph G the following equations hold:

$$\text{daughters}(w) = \bigcup \{l(w) | l \in L\} \quad (1)$$

$$\text{down}(w) = \{u | w \rightarrow^* u\} \quad (2)$$

$$\text{eqdown}(w) = w \cup \text{down}(w) \quad (3)$$

Where the relation $\rightarrow^* \in V \rightarrow V$ is defined such that $u \rightarrow^* v$ if and only if there exists a path in G from u to v .

Finally the set **roots** is defined as the set of nodes without predecessors. I.e.:

$$\text{roots} = \{w | \{u | w \in \text{daughters}(u)\} = \emptyset\} \quad (4)$$

Formalizing condition 1 needs to require that the sets $\text{daughters}(w)$ and $\text{daughters}(w')$ are disjoint for any two distinct $w, w' \in V$, which can be done using the *disjoint union* operator \uplus ¹.

$$V = \text{roots} \uplus \biguplus \{\text{daughters}(w) \mid w \in V\} \quad (5)$$

Condition 2 is a simple cardinality constraint:

$$|\text{roots}| = 1 \quad (6)$$

Condition 3 does not demand any more than that daughters , down and eqdown are well defined in the sense that no node w occurs in its own $\text{down}(w)$. However, one should keep in mind that the $l(w)$ functions are not known beforehand but to be determined. Defining $\text{down}(w)$ with the help of $\rightarrow^*(w)$, which depends on the choice of many $l(w)$ for each w , would lead to the introduction of many extra variables which would result in a very complex CSP. Instead, the recursive definition stated below will be used. Note that termination of the recursion is granted by using \uplus instead of \cup in the first conjunct of (7). This ensures that $\text{down}(w)$ is strictly smaller than $\text{eqdown}(w)$.

$$\begin{aligned} \forall w \in V \\ \text{eqdown}(w) &= \{w\} \uplus \text{down}(w) \\ \wedge \quad \text{down}(w) &= \cup \{\text{eqdown}(w') \mid w' \in \text{daughters}(w)\} \\ \wedge \quad \text{daughters}(w) &= \uplus \{l(w) \mid w \in L\} \end{aligned} \quad (7)$$

The treeness constraints hence consist of a conjunction of (5), (6) and (7).

2.4 Mozart Implementation of the Treeness Constraints

In order to give a more detailed account of the modelling of tree configuration problems, a model implementation of a tree generator in the Oz programming language is presented here. This will give an idea of what variables are used and in which way the constraints can be stated. It is interesting to observe that the implementation of the treeness constraints is merely a straight forward translation of the equations stated above into Oz syntax.

The main data structure of the model implementation is the list `Tree` which has an entry for each possible node index.

```
!Tree = {List.map Indices fun {$ I} {MakeNode I N L} end}
```

The nodes are generated by mapping the function `MakeNode` on the node indices. `MakeNode` initializes a node record containing the node's index (`node`), the sets `daughters`, `down`, and `eqdown` and the list `l daughters`.

¹ $A \uplus B = A \setminus B \cup B \setminus A$

```

fun {MakeNode Index N L}
  unit(
    node      : Index
    daughters : {FS.var.upperBound 1#N}
    down      : {FS.var.upperBound 1#N}
    eqdown    : {FS.var.upperBound 1#N}
    ldaughters : {FS.var.list.upperBound L 1#N}
  )
end

```

Given a node w represented by the variable W , W .`daughters` corresponds to $\text{daughters}(w)$ and $l(w)$ is represented by the l^{th} entry of the list in W .`ldaughters`. Along with `Tree` several additional variables are defined.

```

Roots      = {FS.subset $ NodeSet}
DaughterSets = {List.map Tree fun {$ N} N.daughters end}
EqDowns    = {List.map Tree fun {$ N} N.eqdown end}

```

The finite set variable `Roots` is equivalent to the set `Roots`. `DaughterSets` and `EqDowns` list the daughters and eqdown values of all nodes in the tree.

The first two treeness conditions can then be postulated by translating equation (5) and (6) to Oz syntax.

```

NodeSet = {FS.partition Roots|DaughterSets}
{FS.cardRange 1 1 Roots}

```

The cycle-freeness constraints from equation (7) require to iteratively post constraints on each entry of `Tree`. The auxiliary variable `Self` constitutes a set representation of the node's index. The `Select.union` propagator is not part of the Mozart Oz Base Environment but can be installed with the *Select Constraint Package*². The semantics of the respective line of code in this example is to require that W .`down` contains exactly a union of the elements of the list `EqDowns` whose indices are in the set W .`daughters`, i.e. W .`down` is defined through the eqdown values of the nodes contained in W .`daughters`.

```

for W in Tree do
  Self = {FS.value.singl W.node}
in
  %define eqdown using down
  W.eqdown = {FS.partition [Self W.down]}
  %define down using eqdown of daughters
  W.down = {Select.union EqDowns W.daughters}
  %define daughters using the l(w) sets
  W.daughters = {FS.partition W.laughters}
end

```

² Usually available through the Mogul archive. URI:
x-ozlib://duchier/cp/Select.ozf

Together, these constraints can be put in a procedure `MakeTree` that takes the number of nodes as argument and also features further constraints on the structure of the tree as well as the distribution.

```
{FS.distribute naive DaughterSets}
for W in Tree do
  {FS.distribute naive W.ldaughters}
end
```

`MakeTree` will then bind `Tree` to all trees licensed by these constraints.

2.5 Selection of Lexical Entries

To associate grammatical properties with words, the notion of *lexical entries* has been introduced. A lexical entry carries information like its grammatical *category*, required immediate dependents (*valency*) and syntactic properties like gender, person and number which might decide on the appropriateness of the word for a particular syntactic role (*agreement*). Which lexical entry is chosen follows from how the word is integrated during parsing. It is important that the choice of a lexical entry is made coherently. This makes sure that when the word is integrated into the parse tree according to category, agreement and valency all the decisions taken to this end follow the same interpretation of the word.

Let the functions $\text{cat}(w) : V \rightarrow F_{\text{cat}}$, $\text{val}(w) : V \rightarrow F_{\text{val}}$ and $\text{agr}(w) : V \rightarrow F_{\text{agr}}$ denote the desired feature assignments. Where F_{cat} etc. denote the sets of possible feature values which will not be considered here. Finding the appropriate assignments is hence the task of selecting the right entry from the lexicon. For that, the *selection constraint* can be used.

Selection Constraint. Given a variable X that can be equated with one entry of a sequence $\langle V_1, \dots, V_n \rangle$. The selection constraint requires X to be equal to the I^{th} value of the sequence. Such a choice can be represented by:

$$X = \langle V_1, \dots, V_n \rangle[I] \quad (8)$$

The declarative semantics of such a choice can be compared to array lookup:

$$X = V_I \quad (9)$$

The advantage of the selection constraint is that there is I as an explicit selector. I can then be subject to further constraints so that alternatives can be eliminated through constraint propagation. In a parsing scenario there is a selector variable for each word $w \in V$: $\text{entry}(w) \in \{1, \dots, n\}$ where n is the size of the lexicon. This common selector makes sure that lexical choices are done in a coherent manner:

$$\begin{aligned} \text{cat}(w) &= \langle \text{fin-verb}, \text{noun}, \text{prespart} \rangle[\text{entry}(w)] \\ \text{val}(w) &= \langle \{\text{obj}\}, \{\text{dtr}\}, \{\} \rangle[\text{entry}(w)] \end{aligned}$$

$\text{entry}(w)$ now allows to propagate properties: If it becomes apparent that w cannot be a noun, this restricts the possible values of $\text{entry}(w)$, which will result in eliminating $\{\text{dtr}\}$ from the set of possible values of $\text{val}(w)$.

2.6 Word Order

Word order is crucial when deciding on the grammaticality and dependency structure of an input sentence. Whether or not a noun can be a verb's subject frequently depends on its relative position to that verb.

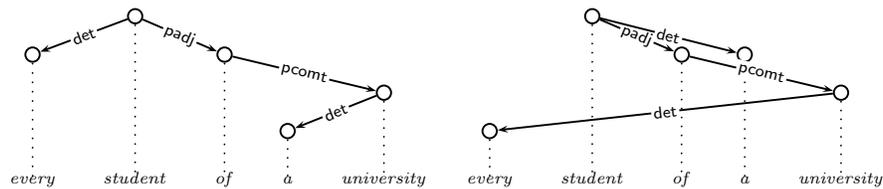


Fig. 3. Two alternative ID trees of the same phrase.

Immediate dominance trees are an instance of labelled trees which are unordered. Consider the two parse trees in figure 3. Clearly the right-hand side ID tree does not describe a valid parse as the determiners are attached in an inappropriate manner. The example suggests to eliminate such trees by demanding the immediate dominance tree to be *projective* when layed out according to the word order. A labelled tree is called projective, if the edges cross neither each other nor the vertical lines underneath a node. In fact, most immediate dominance trees of English sentences are projective as this reflects the natural dependency structure. However, important exceptions can be found in languages like German that allow relatively free word order and when considering grammatical phenomena that lead to re-structuring of phrases like particular forms of questions or raising. Figure 4 shows the parse tree of a German phrase that is valid even though it is non-projective.

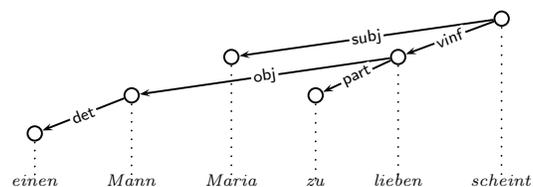


Fig. 4. Non-projective ID tree of a valid German phrase

In order to account for word order while maintaining the required amount of freedom, *topological fields* are identified that allow to describe the relative

positions of words. For any two words w and w' there exists a field of w in which w' is located. Alternations of words within one field are treated as equivalent while assigning a word to a different field can alter the parsing result. On these topological fields a total order is defined. For the sake of this order, each word w itself is also assigned a field of w it is located in. This allows the order to determine whether a particular field is always located before w or after it. When all but one words of a phrase are assigned a field of another word they are located in, this assignment defines a tree structure.

Linear Precedence (LP) Tree. A labelled tree with additional labels for the nodes given by the function $f : V \rightarrow L$. The tree is projective and a total order \prec is given for L . The tree is ordered such that at each node $w \in V$ all outgoing edges occur in the order defined by \prec . This order of the outgoing edges is partial as there might be several edges with the same label. The node w is located between the outgoing edges with a label smaller than $f(w)$ w.r.t. \prec and those with a larger label. As a consequence w has to occur in the input phrase after phrases headed by words connected by edges with smaller labels and after the others.

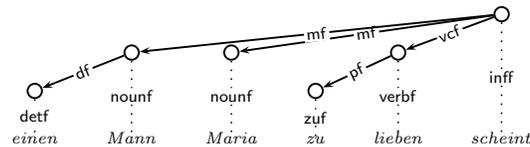


Fig. 5. Linear precedence tree of the phrase of figure 4

The linear precedence tree in figure 5 states that “Maria” and “Mann” are in the *middle field* (mf) of “scheint”, “einen” is located in the *determiner field* (df) of “Mann”. It respects the label order $\text{detf} \prec \text{df} \prec \text{nounf} \prec \text{mf} \prec \text{zuf} \prec \text{pf} \prec \text{verbf} \prec \text{vcf} \prec \text{inff}$.

Such a linear precedence tree can be used in the parsing process in addition to the immediate dominance tree. It is processed in much the same way as described above for the ID tree and underlies constraints in the same way. The lexicon provides information for each word on how it can be integrated in the linear precedence tree similar to valency and category for the immediate dominance tree. Such an additional structure is called a further *dimension*. Dimensions are used to add further types of grammatical information to the grammar formalism like it is done here with linear precedence. It is then possible to allow interactions of this information using *multi-dimensional principles*. These are formed by constraints that restrict the structure of one dimension depending on the structure of the other. Such a multi-dimensional principle is required to connect the *linear precedence dimension (LP)* to the *immediate dominance dimension (ID)*.

This will allow to exclude the invalid parse tree in figure 3 due to its improper treatment of word order.

The principles that govern the relation between the the LP and ID dimension were designed to meet complex grammatical requirements. Carefully comparing the structure of figure 4 and 5 might give an intuition of what is the key idea of the main principle among them. It is called the *climbing principle* and postulates that the LP tree is a flattening of the ID tree. This guaratnees that a phrase headed by a word w can only be the sub-phrase of some phrase headed by w' if w' is in a field of w or one of its parent phrases.

Climbing Principle. There can only be an edge from u to v in the ID tree, if there is an edge to v from u or one of its ancestors in the LP tree.

Note that the ID tree in figure 4 is not projective. However, the word order is legal, because there is a (projective) LP tree (figure 5), that assigns the fields such that these two dimensions follow the climbing principle and can hence be correctly identified as a valid parsing result of the given sentence.

The climbing principle is just one example of how multiple dimensions interact during parsing. Multiple dimensions allow to describe linguistic aspects separated from each other, while multi-dimensional principles form an interface for their interaction. See section 4 for examples of further dimensions.

3 Example Parsing Process

This section gives an overview of how the constraints and principles presented in this contribution interact during a parsing process. One can think of constraint-based parsing as progressively selecting from the space of all possible graphs those that represent a correct parse. Therefore this example process is described by giving for each type of constraints introduced here a candidate parsing result that does not satisfy that constraint and will therefore be excluded.

The constraints are presented starting from the least specific going towards the most specific, which is ususally a good perspective to take on CSPs. However it does not reflect the order in which it actually takes place. Instead, constraint propagation makes sure that at all times information that is derived at one point is available at all the others so that further constraining can take place. The sentence used for this example is:

A student reads a book.

The starting point for applying the constraints is the sequence of words of the input phrase, which will constitute the set V of vertices in the parse tree.

$\begin{bmatrix} \text{str } \text{"a"} \\ \text{cat } \text{det} \\ \text{agr } \text{sg} \\ \text{val } \{\} \end{bmatrix}$	$\begin{bmatrix} \text{str } \text{"student"} \\ \text{cat } \text{n} \\ \text{agr } \text{sg} \\ \text{val } \{\text{det}\} \end{bmatrix}$	$\begin{bmatrix} \text{str } \text{"reads"} \\ \text{cat } \text{vtrans} \\ \text{agr } \text{s} \\ \text{val } \{\text{obj}\} \end{bmatrix}$	$\begin{bmatrix} \text{str } \text{"reads"} \\ \text{cat } \text{vint} \\ \text{agr } \text{sg} \\ \text{val } \{\} \end{bmatrix}$	$\begin{bmatrix} \text{str } \text{"a"} \\ \text{cat } \text{det} \\ \text{agr } \text{sg} \\ \text{val } \{\} \end{bmatrix}$	$\begin{bmatrix} \text{str } \text{"book"} \\ \text{cat } \text{n} \\ \text{agr } \text{sg} \\ \text{val } \{\text{det}\} \end{bmatrix}$
---	---	---	--	---	--

Fig. 6. AVMs of the words in “A student reads a book.”

Lexical Selection. One thing that has to be made sure when lexical entries are chosen for the words is that the entry is appropriate for the input string. When doing so there might still be several lexical entries for each word. The word “reads” is an example of such an ambiguity as it has a transitive (with object) and an intransitive meaning. The set of lexical entries used here is depicted in figure 6 in the form of *attribute value matrices (AVMs)* which display important parts of the grammatical properties of the lexical entries.

Valency. The purpose of valency constraints is to ensure that all relations established between words in the parse tree reflect syntactic roles that are licensed by the grammar. The valency constraints have not been described in this contribution in further detail. The general idea is that the lexicon provides a valency feature for every lexical entry. The valency feature contains a list of syntactic roles that are required to be filled. In the parse tree there has to be exactly one outgoing edge with the appropriate label from a word v for each syntactic role occurring in the valency of v . Similarly there is a second feature ensuring that the incoming edge of each node has an appropriate label. Figure 7 shows a candidate parsing result that will not pass the valency check as an *adj* edge between “reads” and “book” will not be suggested in any reasonable lexicon.

Treeness Constraints. The syntactic analysis done during parsing is only complete if one structure that captures the entire sentence has been derived. Therefore, the treeness constraints exclude candidate results like in figure 8 (left) that do not form trees. This candidate respects valency but clearly does not satisfy the treeness constraint $|\text{roots}| = 1$ and will therefore be excluded.

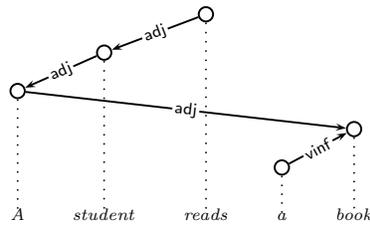


Fig. 7. Candidate parse tree that disrespects valency requirements.

In the same way all parse tree candidates that use the intransitive entry for read will be excluded at this point because only the transitive entry allows an object which in turn is the only possibility to obtain a tree that respects valency and includes both nouns. This is an example of how constraint propagation allows to resolve local (lexical) ambiguities.

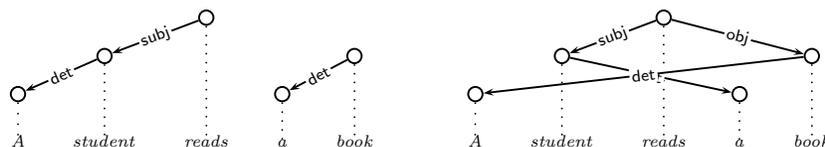


Fig. 8. Candidate ID trees that disrespect the treeness conditions (left) and the word order (right)

Word Order. Figure 8 (right) shows a candidate parse tree which respects valency and treeness but not the order of the words. The word “a” at the beginning of the sentence is assigned the last word, “book”, as a determiner. The determiner of “student” even occurs after this word. To determine whether or not a candidate parse tree respects the word order can be derived with the help of the LP dimension. For each ID tree that belongs to a valid parse, there has to be an LP tree so that the climbing principle holds between them. The LP trees are derived in similar way as outlined here for the immediate dominance trees. For the example of figure 8 (right) the grammar will not admit an LP tree allowing for this configuration: Consider the first “a”. For it to be a determiner of the word “book” at the end of the phrase, there would have to be an edge in the LP to “a” from “book” or from one of its ancestors in the ID tree, i.e. from “reads”. An edge from “book” is not possible as this would contradict the projectivity in the LP tree and from “reads” there cannot be an edge due to rules according to which the LP tree is built: There cannot be an edge from a verb to a determiner.

Result. After excluding all candidates that violate the constraints mentioned here, the ID tree in figure 9 is left. It is displayed along with the respective LP tree that fulfills the climbing principle.

4 Extensible Dependency Grammar

The *Extensible Dependency Grammar* development framework is based on constraint-based parsing. It allows to specify dimensions as well as constraints and multi-dimensional principles. This section presents some dimensions that have been implemented in addition to an ID and an LP dimension along with some of their principles.

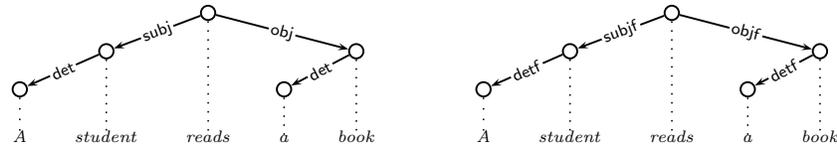


Fig. 9. Correct ID (left) and LP trees (right) for “A student reads a book.”

Deep Syntax Dimension (DS). The deep syntax dimension establishes a representation that is close to the ID representation but can serve as a first step in the direction of semantic analysis. The purpose is to abstract away from some syntactic aspects. For example, in the sentence “By every man a woman is loved.”, “man” is labelled *subj_d* (for deep subject) while it is identified as grammatical object (*obj*) in the ID dimension. As a consequence of such transitions the semantically equivalent sentences “Every man loves a woman.” and “By every man a woman is loved.” have the same deep syntax representation.

Predicate-Argument Dimension (PA). The predicate-argument dimension forms a logic representation that describes the semantics of the input in terms of predicate-variable bindings. Each determiner of the ID dimension introduces a variable that can then be bound as arguments of predicates. For example if in “By every man a woman is loved.” the words “every” and “a” introduce the variables x and y and the word “loved” establishes the predicate *love*, then the binding would be *love*(x, y).

A principle called *linking principle* connects the dimensions DS and PA. It postulates for example, that the first argument of a predicate in PA must correspond to the deep subject in DS so that “every man” would be identified as the first argument which is defined to be a reference to the actor of the action described by the predicate.

Scope Structure Dimension (SC). In order to capture the meaning of a sentence it is important to determine the scope of quantifiers like “a” and “every” in “Every man loves a woman.”. The SC dimension assigns every quantifier a *scope* and a *restriction*. In the respective SC tree, a *restriction* edge would run from “every” to “man”. If one intends to say that for every man there is a particular woman, then there is a *scope* edge from “every” to “a”. One then says that “loves a woman” is in the scope of “every man”. The *contra dominance principle* relates the SC and PA dimension in a way that in order for a noun to be an argument of a verb’s predicate in PA, the verb must be in the scope of the noun in SC.

Quantifier scope is a common source of linguistic ambiguity. In the example phrase, quantification could be in a way that means that all men love the same woman or alternatively in a way that results in the common reading that refers to a less desperate situation. The SC dimension constitutes an elegant way of modelling quantifier scope ambiguity. Parsing the example phrase leads to two

solutions differing only in the SC dimension, which displays two different SC trees for the two readings.

5 Conclusion

This paper presented a way to address the task of parsing natural language using constraint programming. The key idea is to view parsing a sentence as a set of interrelated tree configuration problems. These in turn can be modelled as constraint satisfaction problems with finite set variables. Among other things, constraints postulate, that the resulting graph is a tree, that proper lexical entries are selected for each word and that the parsing result respects grammatical rules like valency as well as the word order of the input phrase.

The advantages of constraint-based dependency parsing over other approaches to parsing lie in constraint programming's inherent capability of treating underspecified information on the desired solution. While allowing to commit to alternative interpretations as late as possible, constraint propagation resolves local ambiguities by integrating information from any partial result as soon as it is derived.

The XDG grammar development framework makes use of constraint-based dependency parsing. The interaction of several dimensions is a key feature of XDG. Multidimensional principles allow to construct representations that abstract away from many grammatical properties present in the immediate dominance dimension. For example, the predicate-argument dimension and the scope structure dimension form the basis of a representation in predicate logic which can be viewed as a first step to semantic processing.

References

- [1] Apt, K. R: Principles of Constraint Programming. Cambridge University Press, 2003
- [2] Debusmann, R., Duchier, D., Kuhlmann, M.: Multi-dimensional Graph Configuration for Natural Language Processing. First International Workshop on Constraint Solving and Language Processing. Roskilde, Denmark, 2004.
- [3] Duchier, D.: Configuration Of Labeled Trees Under Lexicalized Constraints And Principles. Journal of Research on Language and Computation, September 2003