Tutorial of DepPattern

How to write a grammar with DepPattern

January 2009

Contents

Chapter 1

DepPattern: Description of the Formalism

1.1 Basic description

DepPattern is a formalism to write dependency grammars. The DepPattern compiler, called *Compi*, generates robust parsers from DepPattern grammars. The use of the DepPattern compiler is described in the user guide. It has been brought under the GNU General Public License.

A specific DepPattern grammar is constituted by a set of context dependent rules. Every rule is aimed to identify a specific dependent-head relation by means of a pattern of Parts-of-Speech tags. A rule is constituted by two elements:

- a pattern of PoS tags
- the name of a head-dependent relation found within the pattern

Let's see an example:

```
Adjunct : ADJ NOUN
%
```

The first element is "Adjunct", which stands for the name of a dependency relation. Any name can be used for any dependency, only if it was previously declared in the corresponding configuration file (see Section 1.2). The second element is a sequence of PoS tags, which we call "pattern". A pattern must consist of, at least, two tags: the one representing the dependent expression and the other representing the head. Names for tags are declared in the corresponding configuration file (see Section 1.3).

Both, the dependency name and the pattern are separated by two dots (:). Symbol % represents the end of the rule. It is always possible to make use of regular expression operators to tune any character or string : ?,[ ]*, [^], etc.

1.2 Types of dependencies

1.2.1 Main Types

DeptPattern allows a linguist to define the number of dependencies he/she considers they are necessary to build the grammar. If a new dependency is required, he/she must open the configuration file and write a new line with the name of the dependency and its type. DepPattern defines 2 basic types of dependencies:

- A syntactic dependency between two words (a head and a dependent), for instance, the adjunct relation between a noun and an adjective. We call it *open-choice dependency*.
- A lexicon-syntactic relationship between two words, for instance, the relation between the english verb "switch" and the particle "off", which is a syntactic relation giving rise to a new lexical entry, namely the verb "switch off". This type of dependency is called *idiom dependency* of *lexical dependency*. They also produce syntactic relations between lexical units, but, in addition, a new lexical unit is generated by modifying the lemma of the head.

Each basic dependency type has two subtypes according to the relative position of the head and the dependent: the dependent can be either to the left (dependent-head) or to the right of the head (head-dependent). An unlimited number of words can be inserted between both the head and the dependent. So, considering information on word order, the grammar contains 4 different types of dependencies:

DepHead An open-choice dependency where the dependent is to the left of the head. For instance "big monster". The adjective is a left adjunct of the noun.

HeadDep An open-choice dependency where the dependent is to the right of the head. For instance "eat (red) meat". The noun is the direct object appearing to the right of the verb. In this example, there is an inserted adjective between the verb and the noun.

DepHead_lex A lexical dependency where the dependent (or particle) is to the left of the head (or main word). For instance "se arrodilla". The particle "se" is to the left of the reflexive verb.

HeadDep_lex A lexical dependency where the dependent (or particle) is to the right of the head (or main word). For instance "switch (the light) off". The particle "off" is to the right of the verb.

1.2.2 Further types

Dependencies containing a lexical relation

To symplify some linguistic analyses, DepPattern also allows to define dependencies between two words (a head and a dependent) where the syntactic relation between them is lexicalized. In this cases, there is a third gramatical word used as dependency relator. For instance, we can consider that the expression "man with glasses" contains a open-choice dependency between "man" and "glasses" marked by prepostion "with", which is here a kind of binary relator. Obviously, such an expression can also be represented in a more standard way, by means of two basic dependencies ("with" and "glasses", "man" and "with"). Likewise, the expression "if it rains, I go" can contain an open-choice dependency between "rains" and "go", linked by the conjunction "if". These are called *complex open-choice dependencies*.

It would be possible to also find examples of *complex lexical dependencies*. For instance, "have to eat" could be analysed as an idiomatic dependency between "have" and "eat", related by means of particle "to". However, these dependency types are not implemented in the current version of the DepPattern compiler.

The types implemented are the following complex open-choice dependencies:

- DepRelHead
- HeadRelDep
- DepHeadRel
- HeadDepRel
- RelDepHead

- RelHeadDep

Complex dependencies are, in fact, constituted by single binary dependencies. They can be used as syntactic-semantic short-cuts in order to simplify the analysis.

Unary relationship

The types of dependencies described above are used to identify word dependencies. However, DepPattern also permits to identify a PoS tagg in context to make different operations on it: morpho-syntactic corrections, addition of semantic or pragmatic information, modification of some features, etc. For this purpose, we defined a unary relation type: 'Head'. In Section 2.2 of this tutorial, we show an example of how this type of relation is used to solve systematic PoS tagging errors. In addition, in Section 2.3, it is used to set function unicity.

Summary

So, in sum, DepPattern contains 11 types: 2 simple open-choice binary dependencies, 2 simple lexical binary dependencies, 4 complex open-choice binary dependencies, and 1 unary relationship. In further versions, we'll implement the 6 complex lexical dependencies left.

1.2.3 The configuration file: 'dependencies.conf'

Dependency names and their types are declared in the configuration file "dependencies.conf". The number of specific dependencies is open, that is, the user is free to declare the number of dependencies he/she consider being appropriate to define the grammar. Every dependency must belong to one of the 10 types defined above. Each line of the configuration file consists of two columns: the first column contains the name of a dependency, whereas the second column contains its type. Let's see an example:

AdjunctLeft	DepHead
AdjunctRight	HeadDep
SpecifierLeft	DepHead
SpecifierRight	HeadLeft
SubjectLeft	DepHead
SubjectRight	HeadDep
DObjectLeft	DepHead
DObjectRight	HeadDep
PrepComplLeft	DepRelHead
PrepComplRight	HeadRelDep

As word order is involved in the defintion of dependency types, we need to take into account the relative position of the two words (dependent and head) when we associate a name to a specific dependency. In the example above, we do not use simple names such as Adjuncts, Specifiers, Subjects, etc. Each dependency is assigned two complementary names. On the one hand, AdjunctLeft, SpecifierLeft, Subjectleft,... stand for dependencies in which the dependent word occurs at the left position with regard to the head. On the other hand, AdjunctRight, SpecifierRight, SubjectRight,... represent dependencies where the dependent is at the right. However the user is free to choose whatever name for his/her dependencies. The example above is just a proposal. The only requirement the user must fill is to assign only one particular type to each dependency name.

1.3 List of PoS tags and list of morpho-syntactic features

1.3.1 Tagset

The tagset depends on the system used to tag the input text. For instance, the tagset of the English Tree-Tagger is different from that of the Spanish Tree-Tagger, which is different from that of the Spanish Freeling, etc. However, the parser uses as input the output of a tool whose aim is to convert the main PoS tags of all those taggers into a shared list of tags. The shared list is the following: ADJ (adjective), ADV (adverb), NOUN (noun), PRP (preposition), CARD (cardinal number), CONJ (conjunction), DT (determiner), PRO (pronoun), VERB (verb), I (interjection), and 25 more tags for punctuation marks. In addition, there are still some PoS tags belonging to only one tagger. For instance, the English tree-tagger also contains specific tags such as: PoS ('s), PCLE (particle), EX (existential 'there'), etc.

The configuration file where the names of tags are declared is called tagset.conf. Each line contains two columns. The second column contains the names of tags actually used by the system. These names correspond to both the list of PoS tags shared by all PoS taggers, and those PoS tags which are specific to each PoS tagger. The first column shows the names chosen by the user to build the grammar. The user is free to use whatever name. All regular PoS tags are written with upper-case letters. Let's see an example:

```
ADJECTIVE   ADJ
ADVERB      ADV
PREP        PRP
C           CONJ
NUMBER      CARD
DET         DT
NOUN        NOUN
PRON        PRO
V           VERB
INT         I
POS         POS
PCLE        PCLE
```

It is also possible to create short-cuts using regular expressions, such as:

```
X           [A-Z]+
NOTVERB     [^V][^E]+
PUNCT       F[a-z]+
```

Variable X stands for whichever tag name, NOTVERB for whatever tag except those containing the string VE (like VERB), and PUNCT all tags containing the string F followed by some lower-case letters (i.e., punctuation marks). To define more specific shorcuts, we can also use the dysjunction operation "|":

```
NOMINAL   PRON|NOUN
```

Tags of punctuation marks

Finally, the special tags representing punctuation marks are in Table 1.1.

Tag SENT is used to represent the end of a sentence. Three symbols are asigned the tag SENT: "¡", "?" and "!". A sentence is a string between two SENT tags. Patterns are defined within sentences. Up to know, DepPattern does not allow to define rules involved more than one sentence.

.	SENT
?	SENT
!	SENT
¡	Faa
,	Fc
[	Fca
]	Fct
:	Fd
"	Fe
-	Fg
/	Fh
¿	Fia
{	Fla
}	Flt
(	Fpa
)	Fpt
«	Fra
»	Frt
...	Fs
%	Ft
;	Fx
+	Fz
−	Fz
=	Fz

Table 1.1: List of tags representing punctuation marks.

1.3.2 List of morpho-syntactic features

PoS tags are enriched by means of a closed set of morpho-syntactic features. All PoS tags have, at least, three features: "token", "lemma", and "pos" (position). The values of these two features are provided by the PoS tagger given a particular word. For instance, if the word "eggs" was tagged as NOUN in the third position of the sentence, the features "token", "lemma", and "pos" will be assigned the values "eggs", "egg", and "2", respectively (note that the first position is "0"). In addition, each PoS tag has its own set of features. Tables 1.2, 1.3, 1.4, 1.5, 1.6, 1.7, 1.8, 1.9, 1.10, and 1.11, show the specific features (first column) for each tag, including the possible values for each feature (second column). The meaning of each value is described in the third column. To symplify the description, the tables below do not show the null value or "0", which is automatically assigned to a feature when the current PoS tagger does not provide such a specific information.

Unfortunately, many PoS taggers used by the parser provide non-zero values for many features. The exception is Freeling for Spanish, Galician, and Portuguese, which contains all morpho-syntactic information required by the parser.

1.4 Description of Patterns

1.4.1 Basic Patterns

Given a rule, a pattern of PoS tags is a sequence of tags used to identify a specific dependency. A Pattern must fill the following requirements:

- It must contain, at least, those tags that are involved in the dependency: both the head and the dependent. Complex dependencies (HeadRelDep, DepRelHead, etc.) also need a third element: the relator.

features	values	description
type	Q	qualifying
	O	ordinal
degree	A	Aumentative
	S	Superlative
gender	M	masculine
	F	feminine
number	S	singular
	P	plural
function	P	participle

Table 1.2: List of features and values for tag ADJ (adjectives).

features	values	description
type	G	general
	N	negative

Table 1.3: List of features and values for tag ADV (adverbs).

features	values	description
type	D	demonstrative
	P	possessive
	T	interrogative
	E	exclamative
	I	indefinite
	A	article
person	1	first
	2	second
	3	third
gender	M	masculine
	F	feminine
	N	neutral
number	S	singular
	P	plural
	N	invariable
possessor	S	singular
	P	plural

Table 1.4: List of features and values for tag DT (determinants).

features	values	description
type	C	common
	P	proper name
gender	M	masculine
	F	feminine
number	S	singular
	P	plural
person	1	first
	2	second
	3	third

Table 1.5: List of features and values for tag NOUN (nouns).

features	values	description
type	M	main
	A	auxiliary
	S	semiauxiliary
mode	I	indicative
	S	subjunctive
	M	imperative
	N	infinitive
	G	gerund
	P	participle
tense	P	present
	I	imperfect
	F	future
	S	past
	C	conditional
person	1	first
	2	second
	3	third
number	S	singular
	P	plural
	N	invariable
gender	M	masculine
	F	feminine

Table 1.6: List of features and values for tag VERB (verbs).

features	values	description
type	D	demonstrative
	P	personal
	T	interrogative
	E	exclamative
	I	indefinite
	X	possessive
	R	relative
	W	wh-word
person	1	first
	2	second
	3	third
gender	M	masculine
	F	feminine
	N	neutral
number	S	singular
	P	plural
	N	invariable
possessor	S	singular
	P	plural
case	N	nominative
	A	accusative
	D	dative
	O	oblique
politeness	P	polite

Table 1.7: List of features and values for tag PRO (pronouns).

features	values	description
type	C	coordinating
	S	subordinating

Table 1.8: List of features and values for tag CONJ (conjunctions).

features	values	description
(no features)	(no values)	

Table 1.9: List of features and values for tag I (interjections).

features	values	description
type	P	preposition

Table 1.10: List of features and values for tag PRP (prepositions).

features	values	description
gender	M	masculine
	F	feminine
number	S	singular
	P	plural
person	1	first
	2	second
	3	third

Table 1.11: List of features and values for tag CARD (cardinals).

features	values	description
number	S	singular
	P	plural

Table 1.12: List of features and values for tag DATE (date and hour).

- It may contain contextual tags, which will be enclosed in square brackets: [ ].
- Any contextual tag can be tunned with standard wildcards representing optionality, iteration, etc., that is, well known operators used by languages based on regular expressions.

The following examples are Patterns that fill the requirements of DepPattern:

ADJ NOUN
[DT] ADJ NOUN
DT [X]* NOUN
VERB [ADV]* [DT]* [ADJ]* NOUN
VERB [DT]+ NOUN
VERB [DT]? NOUN
NOUN PRP [DT]* [ADJ]* NOUN
- -[DT] ADV VERB
VERB NOUN -[PRP]

The first pattern describes an adjective immediately followed by a noun. Both tags are involved in a simple dependency. The second pattern represents the same situation, but in this case there is a contextual determiner which is not involved in the dependency. The third pattern stands for a simple dependency constituted by a determiner followed by a noun and, optionally, by an unlimited number of different tags between them. The inserted tags are not involved in the dependency: they build the context. Tag X is a shortcut defined in the configuration file (see Section 1.3). The fourth pattern represents a simple dependency between a verb and a noun with three optional tags beween them building the context. The context is constituted by 0 or more adverbs, derterminers and adjectives. The fifth pattern represents the same simple dependency but, in this case, the context is not optional: there must be one or more determiners between the verb and the noun. The sixth pattern is similar to the previous one. The difference is that the contextual determiner is optional: there must be 0 or 1 determiner. The seventh pattern represents a complex syntactic dependency between three elements: a noun, a preposition and another noun. There are two optional contextual tags between the preposition and the second noun: 0 or more determiners and 0 or more adjectives. So, wildcards such as *, +, or ? have their standard meaning in regular expressions. The two last patterns contain negative contexts. The 8th pattern introduces a negative context at the left side of the rule. It matches any ADV VERB combination if only if there is no DT to the left. The last one introduces a negative context at the right side. It matches any VERB NOUN combinaion followed of a tag different from PRP.

1.4.2 Patterns with features

Each PoS tag in a pattern may contain morpho-syntactic and/or lexical information. This information is represented by a feature-value structure, noted as a pair <feature:value>. Let's see some examples:

ADV<lemma:very> ADV
PRO<type:R> VERB

The first pattern represents a dependency between two adverbs, one of them is associated to the lemma "very". This is a lexical restriction. The second pattern represents a dependency between a pronoun and a verb. The pronoun is characterized by means of the type "R(elative)". This is a morpho-syntactic restriction. Both lexical and morpho-syntactic restrictions are represented by means of feature-value structures. In the example above, "lemma" and "type" are features, while "very" and "R" are their specific values.

One of the main advantages of DepPattern is that features are only used when they are necessary. Given a PoS tag, all those features that are not specified in a pattern are considered to have value 0.

1.4.3 Patterns with boolean operators

It is possible to define more complex patterns using boolean operators: |means disjunction (OR), and & means conjunction (AND). Operator | can be used for tags, for features, and for values of features. However, for practical reasons, operator & is only used for features. It cannot be used for feature values because, by definition, two different values of a feature are mutually exclusive. In addition, patterns are not required to use it since they alredy presuppose the meaning of tag conjunction. Let's see some examples:

ADV<lemma:very|quite|rather> ADV|ADJ
PRO<lemma:that&type:Q> VERB
VERB<(mode:S)|(tense:P)> NOUN

The first pattern introduces two "|" operators. The first one represents a disjunction among three possible values ("very", "quite", and "more") of the feature "lemma". The second one is an operator on tags: it allows to choose between either an adverb or an adjective. The second pattern introduces the "&" operator between two features that must be filled simultaneously: to be a relative pronoun (R), and to be lexicalized by means of "that". Finally, in the third pattern, there is a disjunction between two different verbal features. Let's note that disjunctions on features by means of the operator | requires the use of brackets: "(feature1:value1)|(feature2:value2)".

The number of arguments of both | and & is unlimited. When combining the two operators (only with features), & must be always within the scope of |. Below, we show some well formed expressions in DepPattern:

Tag1<(feature1:value1)|(feature2:value2&feature3:value3)>
Tag1<(feature1:value1&feature2:value2)|(feature3:value3)>

Let a, b, and c be 3 features. All possible combination of these 3 features with the 2 boolean operators are represented as follows:

DepPattern representation	**Standard bracketed representation**
$(a)\|(b\&c)$	$(a\|(b\&c))$
$(a\&b)\|(c)$	$((a\&b)\|c)$
$(a\&c)\|(b\&c)$	$((a\|b)\&c)$
$(a\&b)\|(a\&c)$	$(a\&(b\|c))$

The first column shows well-formed DepGrammar expressions while the second one depicts the corresponding expressions using a more compact representation. In the standard representation, brackets are used to delimit the scope of the operator. DepPattern representation is not so compact but is easy to read.

1.5 How rules are applied

As we have said before, a rule must contain, at least, the following elements:

dependency_name : PATTERN
%

A grammar is a list of rules. A rule is applied on a tagged expression (input) if the PATTERN provided by the rule matches a sequence of tags within the expression. The application of a rule consistis in identifying a specific syntactic dependency between two tags (boht the head and dependent) belonging to the pattern.

Rules are applied sequencially in an iterative process. Most rules change the input of the next rules to be applied (this will be described in the following subsections). The process stops when no rule can be applied. However, the linguist can choose an algorithm where iteration is precluded. The parsing algorithm without iteration consists in applying rules sequencially; the process stops when the parser achieves the last rule to be applied. In order to set up the restrictive algorithm without iteration, see the user guide (Section 'Extensions', Subsection 'Precluding Iteration').

1.5.1 Uniqueness Principle

A rule not only identifies a dependency between two words, **it also removes the dependent word from the current expression that is being analysed.** The modified expression will be the input of the following rule. So, rules are applied sequencially and modify the input of the next rule to be applied. Such a modification is justified by the Uniqueness principle of Dependency Grammar.

Let's suppose that we build a simple grammar with the following two rules:

AdjunctLeft : ADJ NOUN
%
SpecLeft : DT NOUN
%

Let's analyse the expression "a beautiful mountain". The input string of the parser would be like this:

a_DT_<...> beautiful_ADJ_<...> mountain_NOUN_<...>

The first rule is applied on this string and finds the "ADJ NOUN" pattern. This finding allows the rule to identify the adjunct dependency between "beautiful" and "mountain". But the rule also removes the adjective (which is the dependent expression) from the string that will be the input the following rule. So, the second rule will be applied on this new input:

a_DT_<...> mountain_NOUN_<...>

It finds the "DT NOUN" pattern and then the dependency between "a" and "mountain". In addition, it removes the determiner from the input string. The head "mountain" is the only expression to be analysed in further applications of rules.

According to the "Uniqueness principle", a dependent word only has one head. So, if we identify a dependency relation containing a dependent word which is no more the head of any word, then it means that we have alredy found all dependencies associated to that word and it can be removed from the search space. The fact of removing one by one the dependents from the input string allows us to reduce in a systematic way the search space, which consists of a huge variety of possible patterns of tags.

Considering Uniqueness, the following constraint is required to write well-formed DepPattern grammars:

Constraint 1: The dependent tag of a rule musn't be involved in further rules. In other words, before writing a rule, we must write before all those rules containing heads instanciated by the dependent tag of the current rule.

1.5.2 Uniqueness Principle and contextual tags

When contextual PoS tags are used to define a pattern, it is necessary to take into account whether the contextual tags are or not syntactically related to the dependent tag of the pattern. For instance, take this rule:

```
AdjunctLeft : [DT]? [ADV]? ADJ NOUN
%
```

The contextual tags, [ADV] and [DT], have significant differences. While [ADV] is syntactically related to the adjective, which is the dependent, [DET] is linked to the noun, the head of the dependency. So, if this rule is applied, it will not be possible to identify later the dependency between the adverb and the adjective, given that the adjective has already been removed from the input string. However, as [DT] is related to the head, the dependency between DT and NOUN can be identfied later by using the corresponding rule.

In order to identify as many dependencies as possible, the grammar should contain rules with contextual tags that are not related to the dependent. Otherwise, some possible dependencies will be missed.

So considering the Uniqueness principle and the contextual tags, the user must take into account the following constraint to build well-formed DepPattern grammars:

Constraint 2: No contextual tag must be related to the dependent tag of a rule.

1.6 Environments without the Uniqueness Principle

Constraints 1 and 2 stated above are directly related to the Uniqueness Principle of the Dependency Grammar. However, according to some linguistic theories (e.g. Word Grammar), such a principle seems to be too strong, since it does not permit to deal with some specific linguistic phenomena. In order to properly analyse such phenomena, it would be useful to write complex rules without taking into account Uniqueness and then the two constraints stated above. DepPattern allows two special environments within which both constraints are not applied: 'blocs of rule' and operator 'NoUniq'

1.6.1 Blocks of rules

The first environment without Uniqueness Principle is called a "Block of rules" (or "Block"). Dependent nodes are only removed at the end of the Block.

The syntax of a Block is the following:

```
Rule-1
NEXT
Rule-2
NEXT
.
.
.
Rule-N
%
```

A Block allows two different tasks: on the one hand, it identifies as many dependencies as rules it contains, and on the other, it removes the dependent tags only after having applied all rules of the Block. So, it identifies the main head of the Block, i.e., the only tag that does not play the role of dependent in any rule. Let's see an example. Take the expression "movie that I see", transformed in the following string:

movie_NOUN_<...> that_PRO_<...> I_PRO_<...> see_VERB_<...>

To analyze this expression, we propose the following Block of 3 rules:

```
DObjectLeft : [NOUN] PRO<type:R> [PRO<type:P>] VERB
NEXT
SubjectLeft : [NOUN] [PRO<type:R>] PRO<type:P> VERB
NEXT
AdjunctRight : NOUN [PRO<type:R>] [PRO<type:P>] VERB
%
```

Each rule identifies a specific dependency. The first one identifies the direct object relation between the relative pronoun "that" and the verb "see". The second rule identifies the personal pronoun "I" as being the subject of the verb. And the third rule links the nominal antecedent ("movie") with the verb of the relative clause ("see"), which is its right adjunct. As the noun is the main head of the Block, the other dependent constituents are removed. The removal of all dependents is only performed at the end of the Bloc. So, there is no kind of removal when processing the previous rules.

The rule where the main head of the Block actually plays the role of head (so, it is not a context tag) must be the last one of the bloc. Otherwise, it would be difficult to identify the main head.

1.6.2 NoUniq environment

If the linguist wishes to define a rule without removing the dependent node, he/she can use the NoUniq operator. For instance:

```
DirectObjectR : VERB NOUN
NoUniq
%
```

This rule does not remove the NOUN tag from the search space.

It is also possible to define a rule where both "head" and "dependent" are removed. For this purpose, the "Remove" operator was defined:

```
DirectObjectR : VERB NOUN
Remove
%
```

This rule removes both the VERB and NOUN tags from the search space.

1.7 More optional operators

Rules can be enriched with 4 additional operators: Recursivity, Agreement, Add, and Inherit.

1.7.1 Recursivity

In some cases, a rule or grammatical structure requires to be applied several times to deal with with recursive expressions such as, for instance, "nice red car". In this expressions the noun "car" is modified by two adjectives in the same way. DepPattern can deal with this phenomenon in several ways: using a block of rules, applying twice the same rule, or using the operator "Recursivity". The 3 possible representations are the following:

Block of rules:

```
AdjunctLeft : [ADJ] ADJ NOUN
NEXT
AdjunctLeft : ADJ [ADJ] NOUN
%
```

Repetition of the same rule:

```
AdjunctLeft : ADJ NOUN
%
AdjunctLeft : ADJ NOUN
%
```

Recursivity operator:

```
AdjunctLeft : ADJ NOUN
Recursivity: 1
%
```

"Recursivity" is specified by numeric values: "1" means that the rule must be applied twice. By default, the value is 0, that is a rule is applied only once. This operator permits to specifiy the number of times a rule can be applied. It allows both a more compact representation and an easier control of rule recursivity.

1.7.2 Agreement

In some cases, both the head and dependent require to share the same values for some of their features. For instance, in Romance languages, the adjective must agree with the noun in both gender and number. Once more, we can represent agreement in different ways. To allow analysing expressions such as "coche rojo" or "coches rojos", and not incorrect ones like "*coche roja" or "*coches rojo", we can make use of a very encumbered representation with 4 rules:

```
AdjunctLeft : ADJ<number:S&gender:M> NOUN<number:S&gender:M>
%
AdjunctLeft : ADJ<number:S&gender:F> NOUN<number:S&gender:F>
%
AdjunctLeft : ADJ<number:P&gender:M> NOUN<number:P&gender:M>
%
AdjunctLeft : ADJ<number:P&gender:F> NOUN<number:P&gender:F>
%
```

However, DepGrammar also allows using the operator "Agreement" which take as arguements the features involved in the agreement operation:

```
AdjunctLeft : ADJ NOUN
Agreement: gender, number
%
```

1.7.3 Add

In some cases, it could be useful to either add a new feature-value to the head or modify the value of one of its features. These two operations can be performed by making use of operator "Add". For instance, the list of morpho-syntactic features used by DepPattern does not contain the verb property "voice". The operator allows to introduce a new feature specifying the voice of the head verb (passive or active) after having applied a grammatical rule. For instance, consider a pattern

identifying a semi-auxiliary verb (dependent) occurring to the left of a past partiple verb, its head (as in the expression "was eaten"). Operator "Add" can be used to assign the passive voice to the head verb:

```
SpecifierLeft: VERB<type:S> VERB<mode:P>
Add: voice:passive
%
```

This new morpho-syntactic information, introduced by a grammatical rule, can be used as the input in further rule applications. Notice that Add can be very useful to correct systematic errors made by the tagger, since it also allows modifying values of existing features.

1.7.4 Inherit

The Inherit operator takes a list of features as input, identifies the values of the dependent expression and assigns them to the corresponding features of the head. That means that this operation allows the head to inherit the values of some features of the dependent. This operation can be used to deal with verbal periphrases. It allows to transfert the morphological properties of a light verb to the content verb (only if the latter is considered the head). For instance, let's assume that there is a rule analysing "had to work" as a dependency between "have" (the dependent) and "work" (the head), via preposition "to" (the relator):

```
PeriphrasisLeft: VERB<lemma:have> PRP<lemma:to> VERB
Inherit: mode, tense
%
```

The Inherit operetor transfers the past tense information from "had" to "work", which is the head of the expression. Let's note that the fact of considering "work" as the head of the dependency is only one of the possible syntactic representations.

1.7.5 Operators within blocks of rules

Recursivity cannot be applied to the individual rules of a block. Instead, it is applied on the whole block when the operator is placed after the last rule.

Agreement can be applied on the individual rules of a block. Concerning the use of this operator, there are no differences between rules within a bloc and regular rules.

Add and Inherit only can be applied on the last rule of a block. The reason is that these operators are aimed to modify the features of a head and a block only returns the main head.

1.7.6 Summary of operators

The list of operators DepPattern allows the linguist to use are the following:

NoUnique It does not remove the dependent tag. It has no arguments

Remove It removes both the head and the dependent. It has no arguments

Recursivity It applies a rule a number N of times before applying the following rule. It has one argument: an integer.

Agreement It requires the dependent and the head to share the same values with regard to a list of features. It has one argument: a list of features.

Add It adds a new feature-value to the head. If the feature already exists, it only modifies the value of the existing feature. It has one argument: a list of feature-value pairs.

Inherit It allows the head to inherit some values from the dependent. It has one argument: a list of features (those from the the head inherit the values).

Corr It enables changing PoS tags using a unary relation (type "Head"). It will described latter in Section 2.2.

1.8 Lexical Classes

DepPattern is also provided with a configuration file, called "lexical_classes.conf", containing word sets likely to be used in any rule. Let's see an example: instead of using directly in a rule all those possible lemmas considered as adverbial quantifiers, it is more economical to declare a lexical class, called for instance "$Quant", instanciated by the corresponding lexical units:

$Quant = very quite more less

$Quant is a lexical variable likeky to be used in whatever rule. For instance, the rule

AdjunctLeft : ADV<lemma:$Quant> ADJ
%

states that there is an adjunct dependency between an adverb belonging to the lexical class $Quant and any adjective at its right.

Any set of words, even a huge list learnt automatically from a corpus, can become a lexical class.

1.9 Begin and end of sentences

To describe patterns taking into account the first tag of sentences, it will be possible to use the restriction <pos:0>. For instance:

SubjL : PRO<pos:0> VERB
%

This rule is applied when the subject pronoun is the first element of the sentence (position = 0).

It is also possible to use symbol " ^ " to represent a head that has no dependent elements to its left (apart from the dependents inserted in the pattern). For instance:

SubjL : ^ NOUN|PRO VERB
%

That means there is only a NOUN or a PRO linked to the verb as a left dependent. There are not other complements or modifiers linked to the verb in the sentence appearing to the left of the nominal subject.

It is also possible to use the attribute "pos" to represent any position of a tag. For instance, <pos:3> represents the forth position of a tag.

Finally, the end of a sentence can be represented by tag SENT. For instance:

SubjR : [PRO<lemma:how>] [ADJ<lemma:old>] VERB<lemma:be> NOUN|PRO SENT<lemma:/?>
%

This rule represents the reverted subject relation within a interrogative sentence such as "how old are you?".

Chapter 2

Examples of Use

2.1 A sample grammar

To give an idea of how a DepPattern grammar can be built, let's propose the following set of rules:

AdjunctRight : VERB ADV
Recursivity: 1
%
AdjunctLeft : ADV VERB
Recursivity: 1
%
AdjunctLeft : ADJ NOUN
Recursivity: 1
Agreement: number, gender
%
AdjunctLeft : DT NOUN
Recursivity: 1
Agreement: number, gender
%
SubjLeft : NOUN|PRO VERB
Agreement: number, person
%
DObjRight : VERB NOUN|PRO
%

This small grammar is able to correctly analyse expressions such as "fast cars", whose output analysis (with flag -a) is the following:

```
SENT::<fast_ADJ_0_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
cars_NOUN_1_<number:P|lemma:car|gender:0|person:3|type:C|token:cars|> ._SENT>

(AdjnL;car_NOUN_1;fast_ADJ_0)
---
Fim do parsing...
```

"a nice fast car":

```
SENT::<a_DT_0_<number:0|lemma:a|possessor:0|gender:0|person:0|type:0|token:a|>
nice_ADJ_1_<number:0|function:0|degree:0|lemma:nice|gender:0|type:0|token:nice|>
fast_ADJ_2_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
car_NOUN_3_<number:S|lemma:car|gender:0|person:3|type:C|token:car|> ._SENT>
```

```
(SpecL;car_NOUN_3;a_DT_0)
(AdjnL;car_NOUN_3;nice_ADJ_1)
(AdjnL;car_NOUN_3;fast_ADJ_2)
---
Fim do parsing...
```

"all the nice fast cars":

```
SENT::<all_DT_0_<number:0|lemma:all|possessor:0|gender:0|person:0|type:0|token:all|>
the_DT_1_<number:0|lemma:the|possessor:0|gender:0|person:0|type:0|token:the|>
nice_ADJ_2_<number:0|function:0|degree:0|lemma:nice|gender:0|type:0|token:nice|>
fast_ADJ_3_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
cars_NOUN_4_<number:P|lemma:car|gender:0|person:3|type:C|token:cars|> ._SENT>

(SpecL;car_NOUN_4;all_DT_0)
(SpecL;car_NOUN_4;the_DT_1)
(AdjnL;car_NOUN_4;nice_ADJ_2)
(AdjnL;car_NOUN_4;fast_ADJ_3)
---
Fim do parsing...
```

"all the fast cars run fast":

```
SENT::<all_DT_0_<number:0|lemma:all|possessor:0|gender:0|person:0|type:0|token:all|>
the_DT_1_<number:0|lemma:the|possessor:0|gender:0|person:0|type:0|token:the|>
nice_ADJ_2_<number:0|function:0|degree:0|lemma:nice|gender:0|type:0|token:nice|>
fast_ADJ_3_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
cars_NOUN_4_<number:P|lemma:car|gender:0|person:3|type:C|token:cars|>
run_VERB_5_<number:0|mode:0|lemma:run|gender:0|tense:0|person:0|type:0|token:run|>
fast_ADV_6_<degree:0|lemma:fast|token:fast|> ._SENT>

(SpecL;car_NOUN_4;all_DT_0)
(SpecL;car_NOUN_4;the_DT_1)
(AdjnL;car_NOUN_4;nice_ADJ_2)
(AdjnL;car_NOUN_4;fast_ADJ_3)
(SubjL;run_VERB_5;car_NOUN_4)
(AdjnR;run_VERB_5;fast_ADV_6)
---
Fim do parsing...
```

"Bill bought a fast car":

```
SENT::<Bill_NOUN_0_<number:S|person:3|type:C|lemma:bill|token:Bill|gender:0|>
bought_VERB_1_<number:0|mode:0|lemma:buy|gender:0|tense:S|person:0|type:0|token:bought|>
a_DT_2_<number:0|lemma:a|possessor:0|gender:0|person:0|type:0|token:a|>
fast_ADJ_3_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
car_NOUN_4_<number:S|lemma:car|gender:0|person:3|type:C|token:car|> ._SENT>

(SubjL;buy_VERB_1;bill_NOUN_0)
(SpecL;car_NOUN_4;a_DT_2)
(AdjnL;car_NOUN_4;fast_ADJ_3)
(DobjR;buy_VERB_1;car_NOUN_4)
---
Fim do parsing...
```

"the rich man bought yesterday a nice fast car":

```
SENT::<the_DT_0_<number:0|lemma:the|possessor:0|gender:0|person:0|type:0|token:the|>
rich_ADJ_1_<number:0|function:0|degree:0|lemma:rich|gender:0|type:0|token:rich|>
man_NOUN_2_<number:S|lemma:man|gender:0|person:3|type:C|token:man|>
bought_VERB_3_<number:0|mode:0|lemma:buy|gender:0|tense:S|person:0|type:0|token:bought|>
yesterday_ADV_4_<degree:0|lemma:yesterday|token:yesterday|>
a_DT_5_<number:0|lemma:a|possessor:0|gender:0|person:0|type:0|token:a|>
nice_ADJ_6_<number:0|function:0|degree:0|lemma:nice|gender:0|type:0|token:nice|>
fast_ADJ_7_<number:0|function:0|degree:0|lemma:fast|gender:0|type:0|token:fast|>
car_NOUN_8_<number:S|lemma:car|gender:0|person:3|type:C|token:car|> ._SENT>

(SpecL;man_NOUN_2;the_DT_0)
(AdjnL;man_NOUN_2;rich_ADJ_1)
(SubjL;buy_VERB_3;man_NOUN_2)
(AdjnR;buy_VERB_3;yesterday_ADV_4)
(SpecL;car_NOUN_8;a_DT_5)
(AdjnL;car_NOUN_8;nice_ADJ_6)
(AdjnL;car_NOUN_8;fast_ADJ_7)
(DobjR;buy_VERB_3;car_NOUN_8)
---
Fim do parsing...
```

"he really wants another car".

```
SENT::<he_PRO_0_<number:0|lemma:he|possessor:0|case:0|gender:0|person:0|politeness:0
|type:P|token:he|>
really_ADV_1_<degree:0|lemma:really|token:really|>
wants_VERB_2_<number:0|mode:0|lemma:want|gender:0|tense:P|person:3|type:0|token:wants|>
another_DT_3_<number:0|lemma:another|possessor:0|gender:0|person:0|type:0|token:another|>
car_NOUN_4_<number:S|lemma:car|gender:0|person:3|type:C|token:car|> ._SENT>

(SubjL;want_VERB_2;he_PRO_0)
(AdjnL;want_VERB_2;really_ADV_1)
(SpecL;car_NOUN_4;another_DT_3)
(DobjR;want_VERB_2;car_NOUN_4)
---
Fim do parsing...
```

Rules can be ordered in different ways, since they fill the basic constraints stated berore in this tutorial. However, to be efficient, a DepPattern grammar should be written by cascades of rules representing linguistic layers or modules. An optimal grammar should contain first rules concerning adverb phrases, then adjective phrases, then nominal phrases, and finally verb phrases.

2.2 Using DepPattern to Correct the PoS Tagged Input Text

DepPattern is provided with tools suited to correct errors of the input PoS tagged text. DepPattern allows a linguist to elabore syntactic rules in order to correct systematic mistakes made by the PoS tagger. For this purpose, we are provided with 3 new elements:

- A new type of dependency, "Head", which represents a unary relation (arity 1). In the default configuration file, dependencies.conf, we declared one type unary relation, called "Single".
- A new operation, "Corr", whose aim is to correct all information associated to a lexical unit: type of PoS tag and morpho-syntactic features. It is similar to the operation "Add". The main difference is that "Corr" allows to change the PoS tag itself.

- A new output format obtained using flag -c. Instead of generating as output the dependency triplets identified by the grammar (flag -a), we can use flag -c to rewrite the same input, but containing all corrections made by operations such as "Corr", or "Inherit", or "Add".

Let's see an example. Suppose that the PoS tagger systematically tag as a subordinate conjunction the word *that* following a noun, even if in this context *that* is, in general, a relative pronoun. To solve the problem, we can write a rule as follows:

```
Single : [NOUN] CONJ<lemma:that&type:S>
Corr: tag:PRO, type:R
%
```

This way, the information introduced by the operator "Corr" is used to change the head expression of the unary relation "Single". It substitutes tag PRO and type R for the information contained in the head (tag CONJ and type S). More precisely, this rule identifies as head a subordinate conjunction with lemma *that* following a noun (its context), and transform this head entry into a relative pronoun. Notice that there there is no dependent expression involved in the rule, since the relation type of "Single" is Head.

"Corr" also allows correcting attibutes by using the values of other attributes:

```
Corr: lemma:=token
```

It means that the value of the lemma is the value of the token. In other words, the lemma attribute inherits the value of the token attribute.

2.3 Function Unicity

The type of dependency "Head" can also be used to take into account the principle of function unicity. This principle states that a verb only contains one main function: one Subject, one Direct Object, and one Indirect Object. So, the grammar should prevent of applying the corresponding rules more than once. To do it, we propose the following strategy. First, we define the following rule:

```
Single : VERB
Add: subj:0, dobj:0, iobj:0
%
```

This means that every verb is provided with 3 new attribute-value pairs (subj:0, dobj:0, and iobj:0), which represent the fact that a verb these 3 functions have not been found yet. Then, all definitions of rules used to identify these functions should contain the following information:

```
SubjL : NOUN VERB<subj:0>
Add: subj:1
%
```

This rule is applied only if the verb has not another subject. Then, the attribute 'subj' is assigned value 1. Then, this rule cannot be applied again.

2.4 DepPattern and Pattern Grammar

DepPattern is a formalism combining notions of both Dependency Grammar and Pattern Grammar.

The main aim of Pattern Grammar is to identify meaningful patterns associated to words. The meaningful patterns of a word can be defined as all the words and structures which are regularly associated with the word and which contribute to its meaning. A meaninful pattern is identified if

a combination of words occurs relatively frequently, if it is dependent on a particular word choice, and if there is a clear meaning associated with it. One of the most relevant assumpations of Pattern Grammar is that there is no a clear boderline between both syntactic and lexical structures.

A very simple formalism is used to represent meaningful patterns of words in Pattern Grammar. For instance, the meaningful patterns of the verb "explain" would be represented as follows:

V n	(explain all the different types)
V wh	(explained how it worked)
V *about* n	(explain about the barman)
V n *to* n	(she explained it to you)
V that	(she explained that she never paid)
V *to* n	(Alex explained to me)
V *to* n that	(have to explain to their patients that they...)

Where **V** stands for the lexical item to be represented (in this case "explain"), symbols **'n', 'wh', 'that'** stands for 'noun group', 'clause introduced by a wh-word', and 'clause introduced by *that*', respectively. Finally, ***to*** and ***about*** are other lexical items being part of a pattern.

DepPattern is provided with the approppriate tools to represent and identify meaningful patterns of lexical words. In order to identify such meaningful patterns in DepPattern, we need to introduce dependency relationships between words instead of phrasal groups. The specific DepPattern rules written to identify the meaningful patterns of "explain" could be the following:

V n	DobjR: VERB<lemma:explain> NOUN
V wh	ObjL: [VERB<lemma:explain>] PRO<type:W> [X]* VERB NEXT DObjR: VERB<lemma:explain> [PRO<type:W>] [X]* VERB
V *about* n	PrepCompR: VERB<lemma:explain> PRP<lemma:about> NOUN
V n *to* n	DobjR: VERB<lemma:explain> NOUN [PRP<lemma:to>] [NOUN] NEXT PrepCompR VERB<lemma:explain> [NOUN] PRP<lemma:to> NOUN
V that	SpecL: [VERB<lemma:explain>] CONJ<lemma:that> [X]* VERB NEXT DObjR VERB<lemma:explain> [CONJ<lemma:that>] [X]* VERB
V *to* n	PrepCompR: VERB<lemma:explain> PRP<lemma:about> NOUN
V *to* n that	SpecL: [VERB<lemma:explain>] [PRP<lemma:to>] [NOUN] CONJ<lemma:that> [X]* VERB NEXT PrepCompR: VERB<lemma:explain> PRP<lemma:to> NOUN [CONJ<lemma:that>] [X]* [VERB] NEXT DobjR: VERB<lemma:explain> [PRP<lemma:to>] [NOUN] [CONJ<lemma:that>] [X]* VERB

These rules should be located at the begining of the verbal phrase layer, and following adverb, adjective, and nominal rules.

Chapter 3

Further Information

3.1 Contributions

Pablo Gamallo Otero and Isaac González
Grupo Processamento da LÃŋngua NaTural (ProLNaT)
University of Santiago de Compostela
Galiza, Spain
pablo.gamallo@usc.es