Chapter 1: Lexicalized PCFG: Parsing Czech

Michael Collins, Lance Ramshaw, Christoph Tillmann, Jan Hajic

Contents

Introduction

Recent work in statistical parsing of English has used lexicalized trees as a representation, and has exploited parameterizations that lead to probabilities directly associated with *dependencies* between pairs of words in the tree structure. Parsed corpora such as the Penn treebank have generally been sets of sentence/tree pairs: typically, hand-coded rules are used to assign head-words to each constituent in the tree, and the dependency structures are then implicit in the tree. In Czech we have dependency annotations, but no tree structures. For parsing Czech we considered a strategy of converting dependency structures in training data to lexicalized trees, then running the parsing algorithms originally developed for English. A few notes about this mapping between trees and dependencies:

- In general, the mapping from dependencies to tree structures is one-to-many: there are many possible trees with a given dependency structure.
- If there are any *crossing dependencies*, then there is no possible tree with that set of dependencies.

Input:
sentence with part of speech tags: I/N saw/V the/D man/N
(N = noun, V = verb, D = determiner)

dependency structure:

word		parent	word number		parent number
I	⇒	saw	1	⇒	2
saw	⇒	START	2	⇒	0
the	⇒	man	3	⇒	4
man	⇒	saw	4	⇒	2

Output: a lexicalized tree

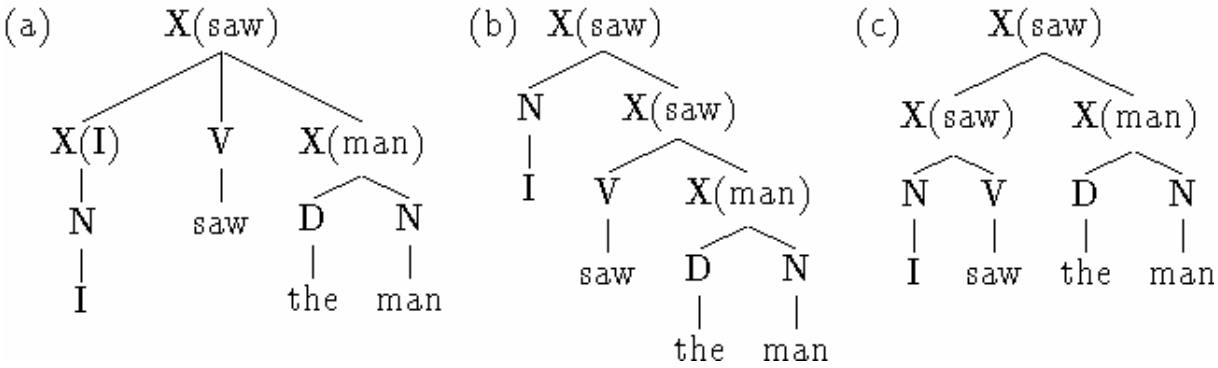


Figure 1: Converting dependency structures to lexicalized trees with equivalent dependencies. The trees (a), (b) and (c) all have the input dependency structure: (a) is the ''flattest'' possible tree; (b) and (c) are binary branching structures. Any labels for the non-terminals (marked *X*) would preserve the dependency structure.

Figure 1 shows an input dependency structure, and 3 lexicalized trees with this dependency structure. We explored different possibilities for the underlying tree structure for the dependency annotations. The choice of tree structure is crucial in that it determines the parameterization of the model: i.e., the independence assumptions that the parsing model makes. There are at least 4 degrees of freedom when deciding on the tree structures:

1. How ''flat'' should the trees be? The trees can be as flat as possible, as in figure 1(a), or binary branching as in trees (b) or (c), or somewhere between these two extremes.
2. What set of part of speech (POS) tags should be used?
3. What non-terminal labels should the internal nodes have? The non-terminals (*X* in trees (a), (b) and (c)) could take any value and still preserve the dependency structure.
4. How should we deal with crossing dependencies?

The Baseline Approach

To provide a baseline result we implemented what is probably the simplest possible conversion scheme, making the following assumptions:

1. The trees were as flat as possible, e.g., as in figure 1(a).
2. The part of speech tags were just the major category for each word (the first letter of the Czech POS set). For example, N = noun, V = verb, etc.
3. The non-terminal labels were ‘‘XP’’, where X is the first letter of the POS tag of the head-word for the constituent. See figure 2 for an example.
4. We ignored the problem of crossing dependencies: effectively any crossing dependencies in training data are dealt with by reordering the words in the string to give non-crossing structures (a natural consequence of the algorithm described later in this paper). The sentences in test data are (of course, for a fair test) left unchanged in order.

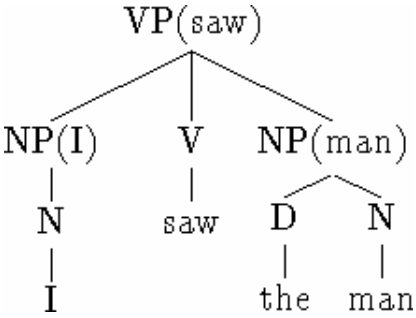


Figure 2: The baseline approach for non-terminal labels. Each label is XP, where X is the POS tag for the head-word of the constituent.

Section 7 gives pseudo-code for the algorithm. This baseline approach gave a result of 71.9% accuracy on the development test set. The next two sections describe a number of modifications to the tree structures that improved the model, and changes to the probability model itself.

Modifications to the baseline trees

Relative Clauses

In the Prague Dependency Treebank (PDT) the verb is taken to be the head of both sentences and relative clauses. Figure 3 illustrates how the baseline transformation method can lead to parsing errors with this style of annotation. The following algorithm was used to modify trees to distinguish relative clauses:

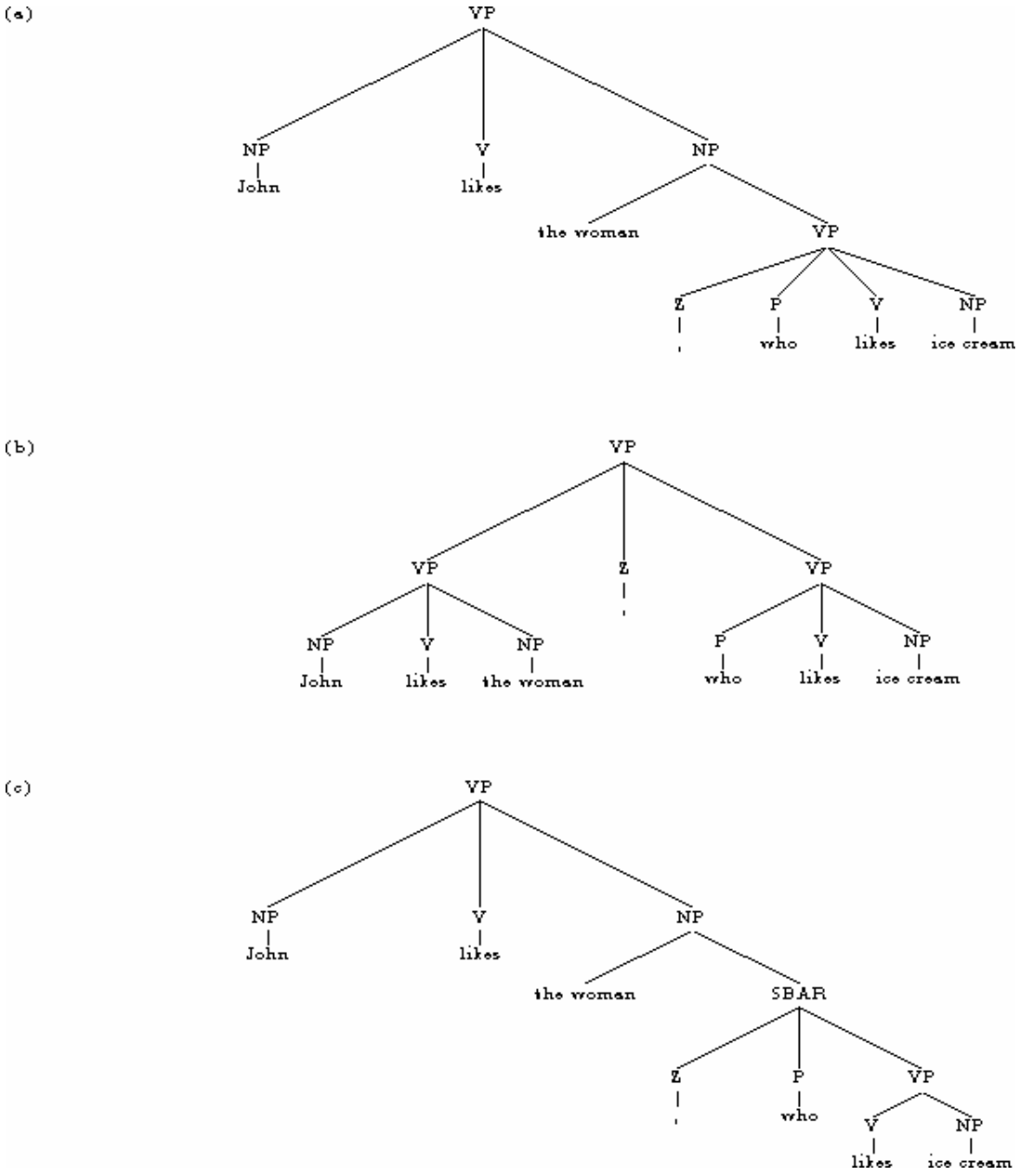
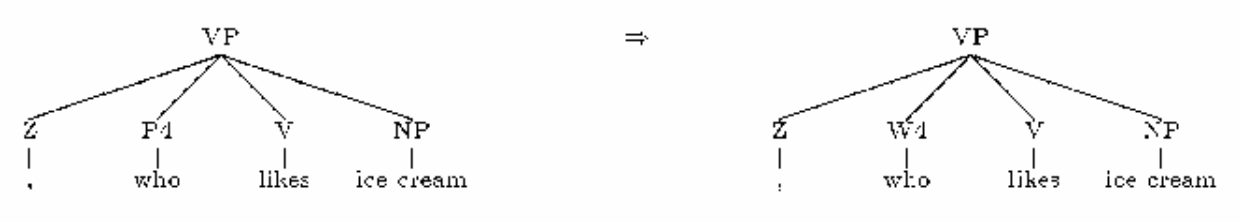


Figure 3: (a) The baseline approach does not distinguish main clauses from relative clauses: both have a verb as the head, so both are labeled VP. (b) A typical parsing error due to relative and main clauses not being distinguished. (note that two *main* clauses can be coordinated by a comma, as in *John likes the woman , the woman likes ice cream*). (c) The solution to the problem: a modification to relative clause structures in training data.

- Change the major POS category for all relative pronouns from **P** to **W** (**W** is a new category; relative pronouns have main POS **P** and sub-POS **1, 4, 9, E, J, K, Q, or Y**). For example,



- For every relative pronoun dominated by a **VP**, with at least one sister to the right:
  1. Add a new non-terminal labelled **VP**, spanning all children to the right of the relative pronoun. e.g.,

  2. Change the label of the phrase dominating the relative pronoun to **SBAR**. e.g.,

Dealing with WH-PPs and WH-NPs

Relative pronouns are not the only signals of relative clauses: prepositional phrases containing relative pronouns (WH-PPs) or noun phrases containing relative pronouns (WH-NPs) can also be important --- see figure 4 for examples.

To account for these relative clauses we applied two stages of processing: 1) find all WHPPs and WHNPs in the training data, and change their non-terminal labels accordingly; 2) apply the relative clause transforms as before but treating WHPPs and WHNPs in the same way as relative pronouns.

Identification of WHPPs and WHNPs is done using the following recursive definitions:

- A WHPP is any PP which has a relative pronoun (POS tag with W as its first letter) as a child.
- A WHNP is any NP which has a relative pronoun (POS tag with W as its first letter) as a child.
- A WHNP is any NP which has a WHNP as a child.
- A WHPP is any PP which has a WHNP as a child.

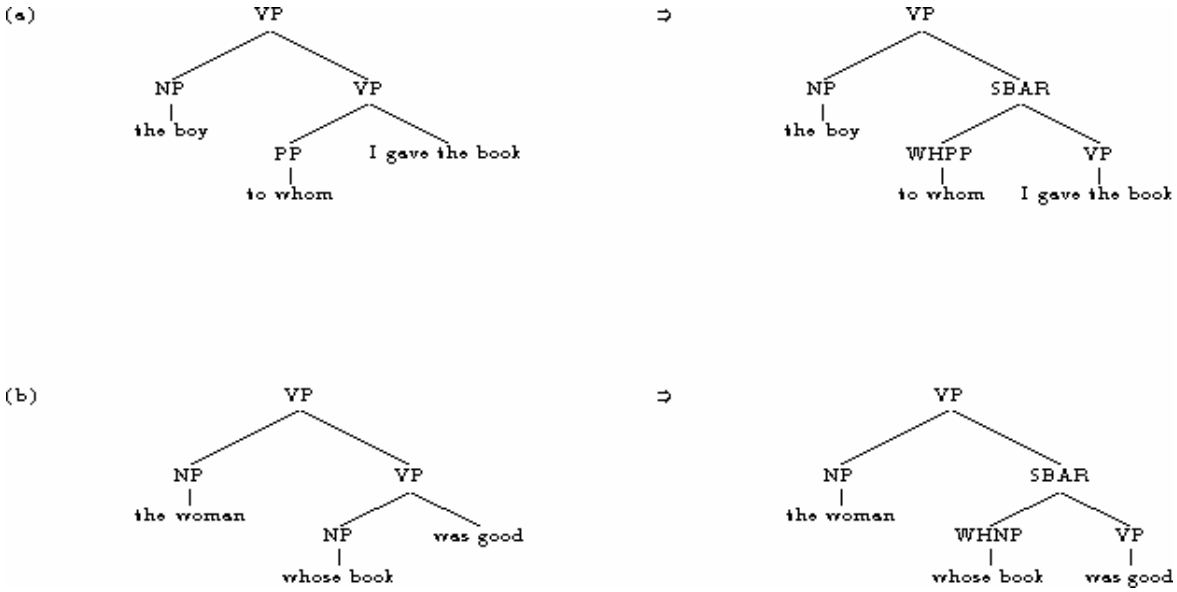


Figure 4: (a) An example of a PP, *to whom*, which has a relative pronoun as its argument. Its non-terminal label is changed to WHPP, and the phrase is changed to reflect it being a relative clause. (b) An example of an NP, *whose book*, which has a relative pronoun as a modifier. Its non-terminal is changed to WHNP, and the relative clause transforms are applied.

Dealing with WH-adverbials

An additional class of relative clause markers are wh-adverbials, for example *where* or *when* (as in *the country (SBAR where it rained)*, or *the day (SBAR when it poured)*. To treat these we identified cases where: 1) a phrase was a VP; 2) the first child of the phrase was a comma; 3) the second child was an adverb (tagged **Db**). A VP level was added covering all children following the **Db**, and the parent non-terminal was labeled **SB**. For example

Dealing with Coordination

To deal with cases of coordination two types of changes were carried out:

- For subtrees of coordination cases the internal labels were changed in order to improve the parameterization of the model (e.g. in Fig. 5, 7).

- New non-terminal labels were used within the tree structure to name new constituents - the orignal flat trees became more structured to improve the parameterization of the model. Examples are given in Fig. 8, 10.

Dealing with cases of coordination improved parsing accuracy by 2.6% as shown in table 3.

Handling of Coordination

When the head of the phrase is a conjunction, the original JP-non-terminal of the phrase is changed using the first letter of the non-terminal of the right-most child (see Fig. 5).

Figure 5: Example for handling conjunction: h_1 and h_2 are the headwords of the left and right NP.

Another change for coordination carried out was to create a new non-terminal based on the first letters of the left-most and the right-most children, e.g. in Fig. 6. After this change the parsing accuracy actually decreased. The most likely reason for that is the large increase in the number of non-terminals. The training data became too sparse for estimating the model parameters.

Figure 6: Example for handling conjunction: h_1 is the headword of the AP, h_2 is the headword of the NP.

When the head of the phrase is a comma, a colon, etc, the original ZP-non-terminal of the phrase is changed using the first letter of the non-terminal of the right-most child (see Fig. 7).

Figure 7: Example for handling punctuation: and are the headwords of the left and right NP.

Figure 7: Example for handling punctuation: h_1 and h_2 are the headwords of the left and right NP.

Handling of Comma

A comma, which was the leftmost-child of a node, but not the head of the phrase, was treated in the following way: an additional non-terminal was added with the comma as the left child and the original phrase as the right child. This proved especially useful for subordinating sentences. The same change was carried out for a comma, which was the rightmost child of a given node, but accuracy was not improved.

Figure 8: Example for handling a comma, which is the left-most child of a phrase: *h* is the headword of the NP. A new non-terminal 'NPX' is introduced.

Extended Handling of Coordination

In the case that within a coordination more than two elements were coordinated, additional structure was added to the tree. An additional non-terminal **XP,** was introduced, where *X* is a variable that can stand for any part of speech of the corresponding head-word, e.g. the tag **'N'** in Fig. 9. Including this change the model was made capable to learn that certain phrases tend to form chains of coordination. Surprisingly, this change did not improve the parsing accuracy.

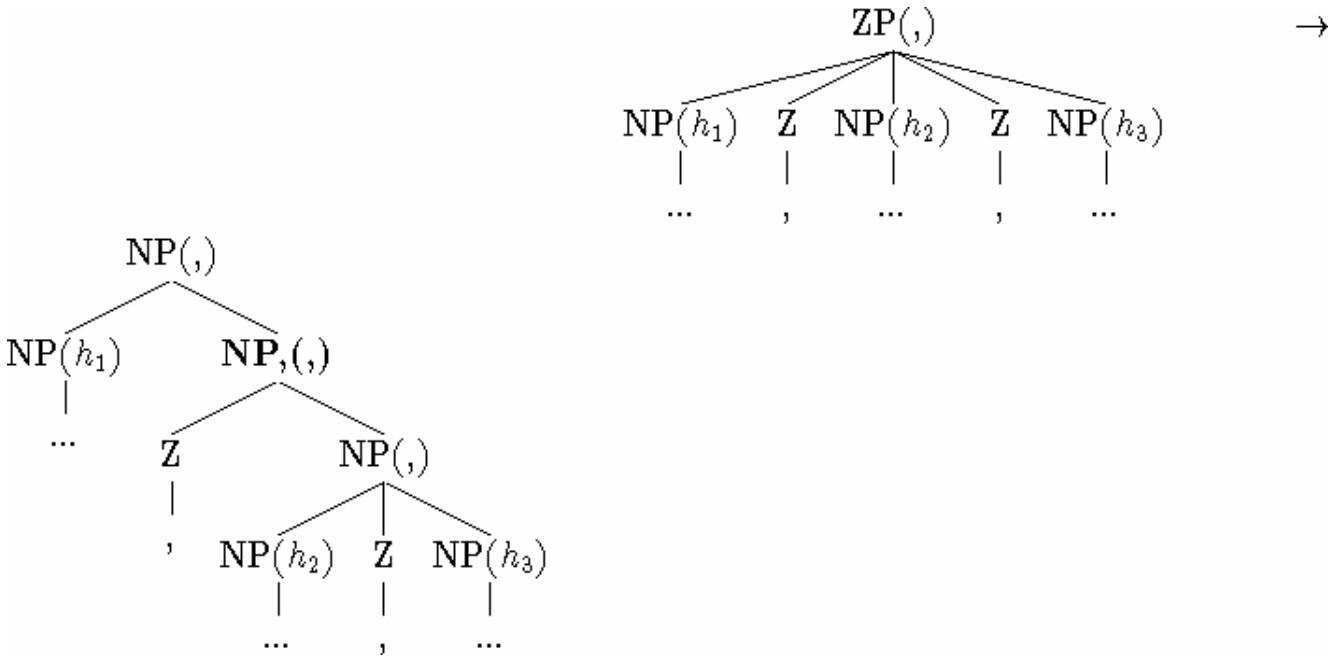


Figure 9: Example of handling several components in coordination: h_1, h_2 and h_3 are the headwords of the NPs. A new non-terminal 'NP,' is introduced.

Handling of Brackets and Quotation

Due to the zero-order assumption for the generation of right/left modifiers, left and right brackets are generated independently of each other. Since the generation process is a zero-markov-order dependency process, the model fails to learn that the pairs should be on the same level of the tree. We introduced a new label, e.g. NP-BRACKETS in Fig. 10, so that a pair of brackets is generated in parallel. For cases of quotation, where the quotes were on the same level of the tree an analogous new label NP-QUOTE was introduced. This change slightly improved accuracy by 0.2 %.

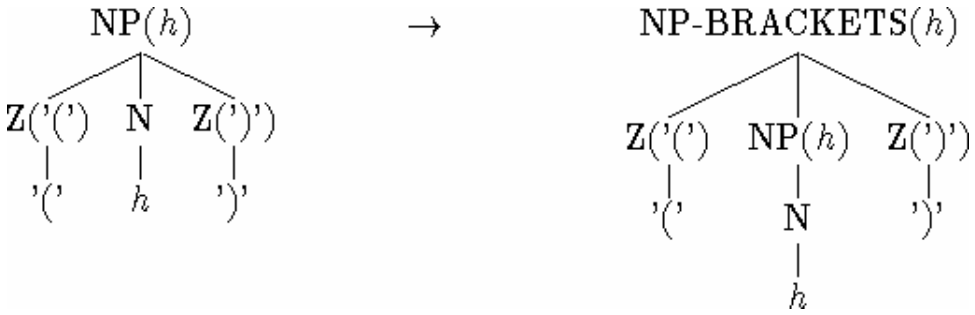


Figure 10: Example for handling brackets: *h* is the head of the NP.

Model Alterations

Preferences for dependencies that do not cross verbs

The models in (Collins 96, 97) had conditioning variables that allowed the model to learn a preference for dependencies which do not cross verbs. We weren't certain that this preference would also be useful in Czech, so we tried the parsing model with and without this condition on the development set (from the results in table 3, this condition improved accuracy by about 0.9% on the development set).

Punctuation for phrasal boundaries

It has been found that in parsing English it is useful to use punctuation (commas, colons, semicolons etc.) as an indication of phrasal boundaries (see section 2.7 of (Collins 96)). The basic idea is that if a constituent #tex2html_wrap_inline551# has two children *X* and *Y* separated by a punctuation mark, then *Y* is generally followed by a punctuation mark or the end of sentence marker. In the parsers in (Collins 96,97), this was used as a hard constraint. In the Czech parser we added a cost of -2.5 (log probability) in the following situation:

- If a constituent X takes another constituent Y as a pre-modifier, and: 1) Y is a comma/colon/semi-colon; 2) the last word of X is not a punctuation mark (tagged **Z**); 3) the word following X is not a punctuation mark; 4) the last word of X is not the last word of the sentence.
- If a constituent Y takes another constituent X as a post-modifier, and conditions 1, 2, 3 and 4 again apply.

Alternative Part-of-Speech Tagsets

Part of speech (POS) tags serve an important role in statistical parsing by providing the model with a level of generalization as to how classes of words tend to behave, what roles they play in sentences, and what other classes they tend to combine with. Statistical parsers of English typically make use of the roughly 50 POS tags used in the UPenn Treebank corpus, but the Czech PDT corpus used for this project provides a much richer set of POS tags, with over 3000 possible tags defined by the tagging system and over 1000 tags actually found in the corpus. Using that large a tagset with a training corpus of only 19,000 sentences would lead to serious sparse data problems. It's also clear that some of the distinctions being made by the tags are more important than others for parsing. We therefore explored different ways of extracting smaller but still maximally informative POS tagsets for use with the Collins parser.

Description of the Czech Tagset

The POS tags in the Czech PDT corpus (Hajic 98) are encoded in 13-character strings. Table 1 shows the role of each character. For example, the tag `NNMP1-----A--` would be used for a word that had ''noun'' as both its main and detailed part of speech, that was masculine, plural, nominative (case 1), and whose negativeness value was ''affirmative''.

1.	Main part of speech
2.	Detailed part of speech
3.	Gender
4.	Number
5.	Case
6.	Possessor’s gender
7.	Possessor’s number
8.	Person
9.	Tense
10.	Degree of comparisson
11.	Negativeness
12.	Voice
13.	Variant / register

Table 1: The 13-character encoding of the Czech POS tags.

Within the corpus, each word was annotated with all of the POS tags that would be possible given its spelling, using the output of a morphological analysis program, and with the single one of those tags that a statistical POS tagging program had predicted to be the correct tag (Hajic and Hladka 98). Table 2 shows a phrase from the corpus, with the alternative and machine-selected tag for each word. In the training portion of the corpus, the correct tag as judged by human annotators was also provided.

Form	**Dictionary Tags**	**Machine Tags**
poslanci	NNMP1-----A-- NNMP5-----A-- NNMP7-----A-- NNMS3-----A-- NNMS6-----A--	NNMP1-----A--
Parlamentu	NNIS2-----A-- NNIS3-----A-- NNIS6-----A-1	NNIS2-----A--
schválili	VpMP---XR-AA-	VpMP---XR-AA-

Table 2: Corpus POS tags for ‘‘the representatives of the Parliament approved’’.

Selection of a More Informative Tagset

In the baseline approach, the first letter, or ;SPMquot;primary part of speech;SPMquot;, of the full POS strings was used as the tag. This resulted in a tagset with 13 possible values. A number of alternative, richer tagsets were explored, using various combinations of character positions from the tag string. Using the second letter, or ‘‘detailed part of speech’’, resulted in a tagset of 60 tags. (The encoding for detailed POS values is a strict refinement of that for primary POS---that is, the possible detailed part of speech values for the different primary parts of speech do not overlap---so using the second letter alone is the same as using the first two letters together.) Combining the first letter with the fifth letter, which

encodes case information, resulted in 48 possible tags. Each of these richer tagsets yielded some improvement in parsing performance when compared to the baseline ‘‘primary part of speech’’ tagset, but a combination of the two approaches did a bit better still The most successful alternative of this sort was a two-letter tag whose first letter was always the primary POS, and whose second letter was the case field if the primary POS was one that displays case, while otherwise the second letter was the detailed POS. (The detailed POS was used for the primary POS values D, J, V, and X; the case field was used for the other possible primary POS values.) This two-letter scheme resulted in 58 tags, and provided about a 1.1% parsing improvement over the baseline on the development set. Even richer tagsets that also included the person, gender, and number values were tested without yielding any further improvement, presumably because the damage from sparse data problems outweighed the value of the additional information present.

Explorations toward Clustered Tagsets

An entirely different approach, rather than searching by hand for effective tagsets, would be to use clustering to derive them automatically, We explored two different methods, bottom-up and top-down, for automatically deriving POS tag sets based on counts of governing and dependent tags extracted from the parse trees that the parser constructs from the training data. Neither tested approach resulted in any improvement in parsing performance compared to the hand-designed ‘‘two letter’’ tagset, but the implemetations of each were still only preliminary, and it might well be that a clustered tagset more adroitly derived could do better. The bottom-up approach to clustering begins with each tag in a class by itself. At each step, two classes are joined so as to maximize the average mutual information of the tag classes, as in the IBM work on ‘‘Class-Based *n*-gram Models of Natural Language’’ (Brown *et al.* 92), except that the mutual information here is calculated not over class bigrams in the text, but over pairs of governing tag and dependent tag, collected from the parse trees in the training set. Given the parsing model, the governing tag is the tag of the head of a consituent, and the dependent tag is that of the head of some subconsituent that is either a pre-modifier or post-modifier. The clustering process could be stopped at various cutoffs to test different sized tagsets. The top-down clustering began instead with a cluster tree of a single node, representing all tags in the same class. In each iteration, one leaf of the tree would be split into subclasses, choosing so as to maximize the total probability over all the parse trees in the training corpus of each governing tag generating its dependent tag time the probability of the dependent tag generating the word that actually occurred. (Thus our formulation of top-down clustering depended on the words in a way that our bottom-up clustering did not.) The algorithm relied on a fixed set of splitting rules, which were applied to each leaf at each step in order to select the best split. The splitting rules that were tested involved either a binary split based on whether a given character position in the tag matched a given value, or an n-ary split into as many children as there were values as a given character position. It is interesting to note that one early binary split found in the top-down clustering was based on position 12, which specified the voice of verbs, which had not been tested as a relevant variable in the hand-designed tagsets. As mentioned above, no clustered tagset was found that outperformed the initial two-letter hand-designed tagset, but this may have been due to problems in the inplementations. The comparative measures of parsing performance may also have been thrown off somewhat by other optimizations made to the parser that depended on the two-letter tags.

Dealing with Tag Ambiguity

One final issue regarding POS tags was how to deal with the ambiguity between possible tags, both in training and test. In the training data, there was a choice between using the output of the POS tagger or the human annotator’s judgment as to the correct tag. In test data, the correct answer was not available,

but the POS tagger output could be used if desired. This turns out to matter only for unknown words, as the parser is designed to do its own tagging, for words that it has seen in training at least 5 times, ignoring any tag supplied with the input. For ‘‘unknown’’ words (seen less than 5 times), the parser can be set either to believe the tag supplied by the POS tagger or to allow equally any of the dictionary-derived possible tags for the word, effectively allowing the parse context to make the choice. (Note that the rich inflectional morphology of Czech leads to a higher rate of ‘‘unknown’’ word forms than would be true in English; in one test, 29.5% of the words in test data were ‘‘unknown’’.) Our tests indicated that if unknown words are treated by believing the POS tagger’s suggestion, then scores are better if the parser is also trained on the POS tagger’s suggestions, rather than on the human annotator’s correct tags. Training on the correct tags results in 1% worse performance. Even though the POS tagger’s tags are less accurate, they are more like what the parser will be using in the test data, and that turns out to be the key point. On the other hand, if the parser allows all possible dictionary tags for unknown words in test material, then it pays to train on the actual correct tags. In initial tests, this combination of training on the correct tags and allowing all dictionary tags for unknown test words somewhat outperformed the alternative of using the POS tagger’s predictions both for training and for unknown test words. When tested with the final version of the parser on the full development set, those two strategies performed at the same level.

First-Order Dependencies

Our parser uses lexicalized-context-free rules of the following type:

$$\mathrm{P(h)} \Rightarrow \mathrm{L_1(l_1)} \ldots \mathrm{L_m(l_m)H(h)R_1(r_1)} \ldots \mathrm{R_n(r_n)}$$

H is the head-child of the phrase, which inherits the head-word *h* from its parent *P*. $L_1 \cdots L_n$ and $R_1 \cdots R_n$ are left and right modifiers of *H*. The generation of the RHS is decomposed into the three steps:

1. Generate the head constituent label *H*
2. Generate the modifiers to the right R_i of the head *H*
3. Generate the modifiers to the left L_i of the head *H*

For the english parser the following independence assumption was made: left and right modifiers were generated by seperate 0^{th} order Markov processes. The generation of a modifier does not depend on the already generated modifiers. The independence assumptions were changed to include bigram dependencies while generating the modifiers to the left and to the right. The generation of a left modifier L_i depends now on the immediately preceeding modifier L_{i-1} (the same is true for right modifiers). The head label *H* is generated as before.

1. Generate head label, with probability

$$\mathcal{P}_{\mathcal{H}}(H \mid P, h)$$

2. Generate left modifiers with probability

$$\prod_{i=1..n} \mathcal{P}_{\mathcal{L}}(L_i(l_i) \mid L_{i-1}, \mathrm{P}, \mathrm{h}, \mathrm{H})$$

3. Generate right modifiers with probability

$$\prod_{i=1..m} \mathcal{P}_{\mathcal{R}}(R_i(r_i) \mid R_{i-1}, \mathrm{P}, \mathrm{h}, \mathrm{H})$$

Due to the introduction of first-order dependencies several changes in the parser code became necessary:

- Parameter-Training: Get the counts to estimate the new bigram probabilities
- Dependency-Modelling: The back-off scheme to handle unseen events is changed
- Parsing-Algorithm: The dynamic programming algorithm for the chart parser is changed

The following lexicalized rule gives an example, for which we show its probability under the **zero-order** assumption and the **first-order** assumption:

`S(bought)` $\Rightarrow$ `NP(yesterday) NP(IBM) VP(bought)`

- Zero-order probability:

$$\begin{aligned}
&\mathcal{P}_{\mathcal{H}}(\mathrm{VP} \mid \mathrm{S}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{NP(IBM)} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{NP(yesterday)} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{STOP} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{R}}(\mathrm{STOP} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought})
\end{aligned}$$

For the first-order assumption the modifier labels *NP* and *STOP* are generated using its immediately preceding modifiers, which are in both cases *NP*'s. *STOP* is a special modifier, which denotes the end of the generation of left/right modifiers.

- First-order probability:

$$\begin{aligned}
&\mathcal{P}_{\mathcal{H}}(\mathrm{VP} \mid \mathrm{S}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{NP(IBM)} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{NP(yesterday)} \mid \mathrm{S}, \mathrm{VP}, \mathit{NP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{L}}(\mathrm{STOP} \mid \mathrm{S}, \mathrm{VP}, \mathit{NP}, \mathrm{bought}) && \times \\
&\mathcal{P}_{\mathcal{R}}(\mathrm{STOP} \mid \mathrm{S}, \mathrm{VP}, \mathrm{bought})
\end{aligned}$$

The smoothing of dependency probabilities is illustrated using the following example. For illustration purposes we use linear interpolation for the smoothing. The actual implementation uses a Backing-Off scheme however. The example distribution is the next to the last term taken from the first-order probability.

$$\mathcal{P}_{\mathcal{L}1}(\mathrm{STOP} \mid \mathrm{S},\ \mathrm{VP},\ \mathrm{NP},\ \mathrm{bought}_V) =$$

$$\lambda_1 \tilde{\mathcal{P}}_{\mathcal{L}1}(\mathrm{STOP} \mid \mathrm{S},\ \mathrm{VP},\ \mathrm{NP},\ \mathrm{bought}_V) +$$

$$\lambda_2 \tilde{\mathcal{P}}_{\mathcal{L}1}(\mathrm{STOP} \mid \mathrm{S},\ \mathrm{VP},\ \mathrm{NP},\ \mathrm{V}) +$$

$$\lambda_3 \tilde{\mathcal{P}}_{\mathcal{L}1}(\mathrm{STOP} \mid \mathrm{S},\ \mathrm{VP},\ \mathrm{NP})$$

$$\lambda_4 \tilde{\mathcal{P}}_{\mathcal{L}1}(\mathrm{STOP} \mid \mathrm{S},\ \mathrm{VP})$$

The *STOP* non-terminal in the preceding rule example is predicted using the head-word 'bought' together with its part-of-speech *V*, the parent-label *S*, the head-label *VP* and the immediately preceding modifier *NP*. The more specific distributions are smoothed using the less specific ones.

Introducing the bigram-dependencies into the parsing model helped to improve parsing accuracy by about 0.9 % as shown in Table 3.

Results

The parser we used was model 1 as described in (Collins 97). We ran three versions over the final test set: the baseline version, the full model with all additions, and the full model with everything but the bigram model. The baseline system on the final test set achieved 72.3% accuracy. The final system achieved 80.0% accuracy: a 7.7% absolute improvement and a 27.8% relative improvement. The development set showed very similar results: a baseline accuracy of 71.9% and a final accuracy of 79.3%. Table 3 shows the relative improvement of each component of the model.

Type of change	Sub-type	Improvement
Tree Modifications	Coordination	+2.6%
	Relative Clauses	+1.5%
	Punctuation	-0.1% ??
	Enriched POS tags	+1.1%
Model Changes	Punctuation	+0.4%
	Verb Crossing	+0.9%
	Bigram	+0.9%
TOTAL	Absolute Change	+7.4%
	Error reduction	26%

Table 3: A breakdown of the results on the development set.

Table 4 shows the results on the development set by genre. It is interesting to see that the performance on newswire text is over 2% better than the averaged performance. The Science section of the development set is considerably harder to parse (presumably because of longer sentences and more open vocabulary).

Genre	Proportion (Sentences)	Proportion (Dependencies)	Accuracy	Difference
Newspaper	50%	44%	81.4%	+2.1%
Business	25%	19%	81.4%	+2.1%
Science	25%	38%	76.0%	-3.3%

Table 4: Breakdown of the results by genre. Note that although the Science section only contributes 25% of the *sentences* in test data, it contains much longer sentences than the other sections and therefore accounts for 38% of the *dependencies* in test data.

Pseudo-code for the baseline conversion algorithm

Input to the conversion algorithm

- n words $w_1 ... w_n$
- n POS tags $t_1 ... t_n$
- An n dimensional array $P[1 ... n]$ where $P[i]$ is the parent of w_i in the dependency structure. We assume word 0 is the start word.
- For each word an ordered list of its children (which can be derived from the array P). *Numchilds*[i] is the number of children for w_i. *Childs*[i][j] is the j'th child for w_i, where the children are in sequential order, i.e. *Childs*[i][1] ;*SPMlt*; *Childs*[i][2] ;*SPMlt*; *Childs*[i][3] ... NB. Childs[0], a list of children for the START word, is always defined also.

For example, for the dependency structure in figure 1:

- $w_1 ... w_4$ = {I, saw, the, man}
- $t_1 ... t_4$ = {N, V, D, N}
- $P[1] ... P[4]$ = {2, 0, 4, 3}
- The childs/numchilds arrays:

i	*Numchilds*[*i*]	*Childs*[*i*]
0	1	{2}
1	0	{}
2	2	{1,4}
3	0	{}
4	1	{3}

Output from the conversion function

The output data structure is a tree. The central data type is a node, i.e. the node of a tree. This a recursive data type that specifies the node attributes, including recursive pointers to its children. The ‘‘tree’’ itself is simply a pointer to the top node in the tree. The **Node** data-type has the following attributes:

- **int type**. This is 0 if the node is internal (a non-terminal or POS tag), 1 if the node is a word.
- **int headword**. The head-word for the node --- if type==1 this is the word itself at this leaf.
- **int Numchilds**. The number of children for the node (by definition, this is 0 if the type==1).
- **Node **Childs**. An array of pointers to the children of the node, in left to right sequential order.
- **Node *Headchild** A pointer to the head-child of the phrase, which must also be in the Childs array.
- **char *label** The non-terminal label for the node.

The conversion function

The recursive function has a prototype

```
Node *create_node(int word)
```

This function takes the index *word*, which can take any value from 0 ... *n*, and returns a pointer to a node which is the top-most constituent with that word as a head (either a POS tag directly dominating the word, or a phrase with the word as its head). To create a tree, simply call tree = create_node(0); This will create a node whose head word is the 0’th (start) word, i.e. the entire tree. The pseudocode is as follows:

```
Node *create_node(int word)
{
   //allocate memory for the POS tag node directly above word
   create(Node tagnode);

   //allocate memory for the word itself
   create(Node wordnode);

   //next create the word node
   wordnode.type = 1;
   wordnode.headword = word;

   //next create the POS tag directly above the word
   tagnode.label     = t_word;
   tagnode.type      = 0;
   tagnode.headword  = word;
   tagnode.headchild = wordnode;
```

```
   tagnode.numchilds = 1;
   tagnode.childs[0] = wordnode;

   //if the word has no dependent children in the dependency structure,
   //then just return the POS tag as the node for this word
   //(this will happen for words ‘‘I’’ and ‘‘the’’ in the figure)
   if(numchilds[word] == 0)
      return &tagnode;

   //otherwise we’ll create another node above the POS tag
   create(Node phrasenode);

   phrasenode.label    = t_word + ’’P’’;
   phrasenode.type     = 0;
   phrasenode.headword = word;
   phrasenode.headchild = tagnode;

   //Note the node has as 1 more child than the word has dependents,
   //as there is an extra child for the head of the phrase, the POS
   //tag for the word
   phrasenode.numchilds = numchilds[word]+1;

   //now recursively build the children
   //
   //the one subtlety is that the head of the phrase, the tagnode
   //that we have just built, has to be inserted at the right place
   //in the childs sequence

   n = 0;
   //flag = 0 indicates the head of the phrase has not been inserted
   //yet
   flag = 0;

   for(i=1;i <= numchilds[word];i++)
   {
      if(flag == 0 &&
         word < i)
      {
         //insert the head node
         phrasenode.childs[n] = tagnode;
         n++;
         flag = 1;
      }

      //recursively create the sub-tree for the i’th dependent,
      //and put it in the phrasenode.childs array
      phrasenode.childs[n] = create_node(childs[word][i]);
   }

   if(flag == 0)
   {
      //insert the head node at the end of the phrase
      phrasenode.childs[n] = tagnode;
   }

   return &phrasenode;
}
```

References

1. [Brown et al. 1992]
   Peter Brown et al.: *Class-Based n-gram Models of Natural Language*, Computational Linguistics, Vol. 18, No. 4, pp. 467--479, 1993.
2. [Collins 1996]
   Michael Collins: *A New Statistical Parser Based on Bigram Lexical Dependencies*. In: Proceedings of the 34th Annual Meeting of the ACL, Santa Cruz 1996.
3. [Collins 1997]
   Michael Collins: *Three Generative, Lexicalised Models for Statistical Parsing*. In: Proceedings of the 35th Annual Meeting of the ACL, Madrid 1997.
4. [Hajic 1998]
   Jan Hajic: *Building a Syntactically Annotated Corpus: The Prague Dependency Treebank*. In: Issues of Valency and Meaning, pp. 106-132 Karolinum, Charles University Press, Prague, 1998.
5. [Hajic, Hladka 1998]
   Jan Hajic and Barbora Hladka: *Tagging Inflective Languages: Prediction of Morphological Categories for a Rich, Structured Tagset*, In: Proceedings of ACL/Coling'98, Montreal, Canada, Aug. 5-9, pp. 483-490, 1998.