Rule-based Morphological Disambiguation

Toward a Combination of Linguistic and Stochastic Methods

Pavel Květoň

Abstract

This thesis describes a rule-based morphological disambiguation (tagging) of a natural language (in particular, Czech) which could either replace stochastic taggers, or at least cooperate with them.

The thesis initially deals with *negative n-grams* — simple regular expressions that are able to determine grammatical correctness of sequences of morphological tags. However, negative n-grams are unable to cover all the phenomena in the natural language under investigation. Hence the thesis extends the idea of negative n-grams and introduces general *disambiguation rules*. The rules are formally defined and important properties of them are presented.

A new formalism *LanGR* (the Language for Grammatical Rules) has been developed to implement the disambiguation rules. The thesis also contains tutorial lessons in programming in the *LanGR* formalism.

The fundamental rules describing the Czech language have been already written in *LanGR* and the set of rules is still being extended. The rules have been used in several ways — among others, the rules have performed well in the verification of the manual disambiguation of the Prague Dependency Treebank (PDT). The results of the morphological disambiguation of PDT by the rules are also presented.

Contents

Chapter 1

Introduction

Since the sunrise of the age of computer there have been trials to make computers understand, speak or generally somehow *model* natural languages. In all approaches to the modeling, it early became clear that no good model can be built without large banks of "computerized" texts of a particular natural language — it holds both for the models based on parameters automatically retrieved from sample texts and for the models based on the entire language system where competent people usually need to verify their ideas on parole. And reasonably "computerized" texts offer this necessary feature of searching through in quantities of texts.

In the very beginning of this thesis it should be pointed out that it is strongly focused on the *written* language. It can be assumed that spoken language is much more difficult to recognize because of its "ungrammaticality".

The sample texts are usually referred to as natural language *corpora*. A value of a corpus can be measured in several ways, e.g. by its content (how the content covers the natural language at hand) or by its size (it always holds that a larger corpus is better than a smaller one with the same features). But the most valued feature of every corpus is *annotation* — enriching the plain texts in linguistic information.

There exist various kinds of information that can be added to the plain text — from segmenting the text into separate sentences over morphological analysis and syntactic structures even to semantics and pragmatics. It can be realized in the future that non-linguistic information (e.g. the temperature at the time of the issue of the text) is also valuable. This thesis, however, concerns almost exclusively *morphological* annotation (with the hidden knowledge of syntax, semantics,...).

In the process of adding (e.g. morphological) information to the raw text, it is necessary to distinguish between two steps of the assignment of such information — *analysis* and *disambiguation*:

- the analysis (on any level) is mostly understood as assigning all values that are possible for the particular piece of text without looking at any context;
- the disambiguation means selection of values that are the only correct ones by using any knowledge available (even pragmatics).

1.1 Morphological analysis

On the morphological level, the task of the analysis is to assign *morphological tags* to every single *word* of the input text — each tag is a set of values of morphological *categories*, e.g. number, gender or person, see [22]. For example, the word `hledala` is assigned the following tag:

```
Part-of-speech  = verb
Number          = singular
Gender          = feminine
Person          = third
```

However, without any further knowledge it is also correct to assign a different tag:

```
Part-of-speech  = verb
Number          = plural
Gender          = neuter
Person          = third
```

Thus, morphological analysis assigns all possible tags to a word form (at least two different tags for the verb `hledala`).

In the morphological analyzer, the set of categories that are assigned to the words is fixed. However, it is clear that some categories are irrelevant for some words (e.g. case is irrelevant for verbs). Morphological categories that are irrelevant for a particular word take the special value *n/a* (not applicable). Since domains of the categories are also known, it is possible to create a set of all valid tags that can be assigned by the analyzer at hand — this set is mostly referred to as the *tagset*. It is clear that the analyzer assigns to each word form of the input text a subset of the tagset.

The morphological analyzer itself is not described in this thesis. For the processing of Czech, the analyzer written by Jan Hajič (see [20, 21, 22]) has been used but the overall strategy of this thesis is to be tagset-independent.

Hajič's tagset contains more than 4300 tags (only 1227 tags are really used in the corpus SYN2000, see Section 1.4). The tags are *positional* — it means that all tags are strings that have the same length (15 letters, in this particular case; two letters are unused in the actual version of the analyzer) and each position of a tag is reserved for a single grammatical (morphological) category. For example, let's investigate the tag `AAIS1----1N---9`. The first position denotes the part-of-speech category (`A` stands for Adjective), the second one is the sub-PoS category, the third one is the gender category (`I` for masculine Inanimate)... The full description of the tagset can be found in [14, 22].

In the rest of the thesis, let the abbreviation *MAJH analyzer* denote Hajič's morphological analyzer and let *MAJH tagset* denote the tagset used by the MAJH analyzer.

To improve the operation of the rules, it was necessary to modify the output of the MAJH analyzer. The list of major modifications of the MAJH analyzer follows:

- one of the two unused positions in the tags of the MAJH tagset is used for a syntactic Part-of-Speech category (`SyntPoS`); the value of this category is inserted to every particular tag from the MAJH tagset using a simple algorithm (taking into consideration mainly the Part-of-Speech and sub-PoS categories of the original tag);

- the second of the two unused positions in the tags of the MAJH tagset is used for a clause separator flag (`clsep`); its value is applicable only for conjunctions and punctuation and every such tag from the MAJH tagset is split into two new tags — one with `clsep` set on and one with `clsep` set off;

- the MAJH tagset frequently uses "shortcuts" in tags — a single tag represents, in fact, a set of tags. For example, the `X` value on the position of the case category implies that the tag is equal to the set of seven tags with the case category filled by the particular values `1`–`7`[1];

  The "shortcuts", however, represent a problem for the disambiguation rules. It was decided that the simplest solution of the problems that are caused by the "shortcuts" is to replace every "shortcut" by the set of tags it represents. This modification leads to removal of all "shortcuts" from the MAJH tagset;

- another serious problem is related to digital numbers — in the MAJH tagset, all digital numbers are assigned the single tag `C=----------`. The tag doesn't contain information about case, gender, number and other information that is critical for correct operation of the rules. It was necessary to develop an algorithm that replaces the tag by a set of tags that better meet the requirements of the rules. In a few words: every digital numeral is assigned exactly all tags that are assigned to its written numeral equivalent.

The changes that have been made to the MAJH tagset are serious. In the rest of the thesis, the modified MAJH analyzer (MAJH tagset) is referred to as *E-MAJH* analyzer (*E-MAJH* tagset) and it is assumed that the input of the rules is processed by the E-MAJH analyzer.

1.2 The task of morphological disambiguation

A morphological analyzer assigns each word all morphologically legal interpretations. But what is really expected in the corpus is, in the vast majority of cases, only one tag (the only correct one) for every word. The process of selecting the "correct" tag is known as *disambiguation* or *tagging*. The disambiguation can be performed either manually, or automatically.

Theoretically, people can correctly manually annotate any corpus. In practice, the manual annotation is limited in at least two ways — the corpora cur-

[1] Note that `X` value (and a few other "shortcuts") is a valid value for the particular category and it is different from the *not applicable* value.

rently required are too large to be annotated manually and also the annotators are mostly tired and error-making humans.

Automatic disambiguation is usually quick enough to process large texts. However, at present it still makes more mistakes than competent humans (and any other result would be rather surprising). The problem is that the disambiguation algorithm (mostly called *tagger*) must have "in mind" the entire language system (langue) to achieve the same results.

It is often useful to talk about *partial* disambiguation — it makes it possible to keep more tags within a word whenever it is not possible to select the only one.

1.3 Methods of morphological disambiguation

It has already been said that there are several ways to perform automatic morphological disambiguation. All the methods commonly used today can be divided into three big sets: *stochastic*, *rule-based* and their *combinations*.

Among all purely stochastic methods there are *HMMs* (Hidden Markov Models) and *smoothed n-grams*. Both methods are based on searching the most probable sequence of tags by predicting a tag in dependence of (ambiguous or non-ambiguous) *fixed* context (usually two preceding words at most). The methods are simple to implement. They very quickly process the input corpus. Without any change of the tagging algorithm, these methods can be used to model any natural language whenever the (annotated) training data are available for the language. Even with a very limited context the methods give acceptable results (see [1], [18]).

On the other hand, the errors of these methods are hard to trace and it is also difficult to predict whether more training data lead to a lower number of errors — and more training data should be manually annotated[2] and this is always a problem.

There are two features of stochastic methods that are criticized by linguists:

- firstly, the stochastic methods use limited context, which makes it impossible for them to correctly handle some types of grammatical features (e.g. whenever the words participating in the feature are separated by a lot of other tokens — for example, agreement of subject and predicate that are separated by a subordinated clause);

- secondly, the stochastic methods use cross entropy as a measure of their quality; while the mathematical background of entropy is very strong, the linguists simply object that the (even-more usually approximated due to computational limits) probability of sequence of tags cannot be a good measure of grammatical correctness of a sentence.

While the limited context is a feature that can be somehow avoided (e.g. by more complex stochastic methods, see below), the entropy as a quality measure is used by *every* stochastic method as its fundamental component.

[2] Confer, for instance, the error analysis of the NEGRA corpus of German, see [4].

It is beyond the framework of this thesis to determine whether using the entropy measure can be replaced by another (more grammatical) measure of correctness. In all cases, from the stochastic point of view the entropy is the most correct measure for modeling a sequence of anything (in particular, a sequence of tags).

There are many ways to increase linguistic competence of stochastic methods. In general, combinations of various methods can be divided into two groups: combining standalone rule-based methods with a standalone stochastic tagger and merging linguistic competence directly with the body of a stochastic tagger.

Incorporating the system of language directly into the body of a stochastic tagger requires special taggers. One of such taggers is a *maximum entropy* tagger (see [6, 23]). There also exist models that are based on linguistic rules equipped with some "probabilities" of their "accuracy". These methods are usually known as *probabilistic grammars*, see e.g. [11, 24].

Roughly speaking, a maximum entropy tagger employs so-called "features" — every feature specifies whether a tag is "grammatically" correct or not in the given context (which is theoretically unlimited). In the training stage, the frequencies of occurrences of the features are obtained from the training data (manually annotated corpus) — the model is built to keep these frequencies. With these restrictions, a "weight" is computed for every feature to maximize conditional entropy of the model (i.e. the model is as uniform as it can be with fixed frequencies of the features). In the end, the final subset of features is selected from the set of all features so that it minimizes cross entropy on the training data[3].

Since there is no restriction on the features themselves, the basis of the tagger is completely different from HMM taggers. Unlike HMM taggers, the maximum entropy tagger is very hard to train (i.e. to obtain the weights of the features). It is not known whether any approximation of weights can be as good as the precisely computed weights.

In the probabilistic grammars (usually describing the syntax level of a modeled language), each rule in the grammar is assigned its "weight" — there are several ways of employing and using these weights to process the language. The weights of the rules are retrieved from the training data (i.e. a manually annotated and/or morphologically analyzed corpus or even a raw text). In this context, maximum entropy taggers could also be put under the term of probabilistic grammars.

Unlike HMM taggers, these "hybrid" taggers are more or less dependent on the particular natural language (this dependency is encoded in the construction of features or rules).

As humans manually annotate corpora, they can also use their linguistic competence to write down grammatical rules they use in understanding or validating the texts of a particular natural language. One of the pure rule-based approaches to some special tasks of syntactic analysis (that can be used for the disambiguation on the morphological level) has been presented in the thesis

[3]The subset selection is not necessary in case the total amount of features is not very high.

[35].

While the linguists are able to explicitly state what is correct or incorrect in the natural language, it is wrong to force linguists to assign some weights to their rules (i.e. if some rules are not 100% safe, they could be still used with some level of uncertainty), since nobody can exactly say that "this" rule is "95.13%" safe. These weights are again retrieved automatically from the training corpus — and we are back in the probabilistic grammars.

The weak points of pure linguistic grammars without any probabilities are obvious:

- the approach is strongly language-dependent (grammatical rules cannot be simply applied to another language),
- "full" disambiguation of every input sentence cannot be achieved (i.e. it very often ends up with a partial disambiguation because no set of rules can describe the whole grammar) and
- highly skilled, precisely thinking linguists are required to write the rules.

On the other hand, the advantages are also obvious: it requires no training data (however, some manually annotated data are useful for debugging the written rules), it can be easily traced and it can be used to verify manual annotation or even to check the grammaticality of input texts.

Exactly this kind of rule-based tagging is investigated in this thesis.

1.4 Annotated corpora of Czech

In this section, a brief summary of existing well-known annotated corpora of written Czech is given:

CNC (Czech National Corpus, see [26]) is a corpus of Czech assembled mostly from written Czech texts. It is being maintained by the *Institute of the Czech National Corpus* at the Faculty of Arts, Charles University. Its official (publicly available) part *SYN2000* contains about 100 million running words. The corpus is morphologically analyzed by the analyzer developed by Jan Hajič, see [22]. The tagset of the analyzer is briefly introduced in Section 1.1. The corpus is disambiguated by a stochastic tagger (see [19, 20, 21]).

PDT (Prague Dependency Treebank, see [14]) is a corpus of Czech texts which constitute the part of SYN2000. It is being maintained by the *Institute of Formal and Applied Linguistics* and the *Center for Computational Linguistics* at the Faculty of Mathematics and Physics, Charles University, Prague. It contains about 1,500,000 running words (including punctuation) — the corpus is manually morphologically annotated. It contains syntactic (and partially tectogrammatic) annotation.

DESAM (see [34]) is a corpus of Czech with more than one million running words. It is being maintained by the *Faculty of Informatics*, Masaryk

University, Brno. The morphological analyzer *LEMMA* (see [36]) has been used to obtain the full morphological analysis of the data. The tagset of *LEMMA* is different from the tagset used in CNC — both in the format of tags (*LEMMA* uses non-positional tags) and also in the grammatical categories and their values. The conversion between the two tagsets is possible, but non-trivial.

1.5 Historical note

The idea of rule-based models of Czech is not original — the attempts at formalizing the syntax of Czech were already made in the past (for example, see [17]).

At that time, however, the overall performance of computers was too weak to fit the requirements of the linguists. With increasing computational power, it is possible to process larger corpora. Unfortunately, the idea of formalizing Czech syntax has been put forward by stochastic models — at least because the stochastic models are usually simpler to implement.

The stochastic taggers of Czech actually have an error-rate of about 5%. While the error-rate of 5% seems very low, the performance of the stochastic models in annotating Czech corpora has been criticized by linguists: stochastic taggers systematically fail in several important (and simple to describe) phenomena of Czech grammar.

Thus, in the very end of the last century, a group of linguists was established with the primary objective to improve the annotation of CNC.

The first set of rules (about 80 rules) was written by linguists on sheets of paper and manually encoded in C++ in order to prove the usability of the rules (see [1]). The number of rules was high enough to see several important features of the rules.

The relevance of grammatical rules has been unambiguously proved. It was also clear that the rules should be executable automatically, without their manual encoding into any programming language.

With these baselines the group of mathematical linguists has begun to develop a formalism that would fit all their goals — the formalism *LanGR*.

1.6 The layout of the thesis

Chapter 2 defines the terms used throughout the thesis.

Chapter 3 concerns the task of partial disambiguation in general and points out the goals set by the linguists when the work on the formalism *LanGR* started.

Chapter 4 investigates restricted regular expressions — so-called *negative n-grams* — that are able to determine incorrect sequences of tags. The idea was first introduced by Karel Oliva, see [15]. Formal mathematical framework of negative n-grams has been analyzed by Karel Oliva, Pavel Květoň and Roman Ondruška, see [4, 5].

Chapter 5 defines mathematical framework of disambiguation rules and describes the *LanGR* formalism that makes it possible for linguists to write down the disambiguation rules. The chapter contains a handbook for linguists that would like to write the rules in *LanGR* — the formalism itself has been designed by Pavel Květoň with a lot of comments by Vladimír Petkevič and Karel Oliva. The tutorial rules have been (with minimal changes) taken from the website where the disambiguation rules for Czech are stored (see Appendix A) — the rules themselves have been written by Tomáš Jelínek, Karel Oliva and Vladimír Petkevič with implementation hints of Pavel Květoň.

Chapter 6 concerns implementation issues — interpreting the formalism *LanGR* and compiling the rules using so-called *fast LanGR* into an executable file. The implementation of *LanGR* has been written by Pavel Květoň (see [2]).

Chapter 7 summarizes the results of application of the rules to corpora.

Chapter 2

Basic terms

Several basic notations will be introduced here (see [46] or [47]):

$|M|$ denotes the size (number of elements) of a set M;

$\mathbb{N}$ denotes the set of natural numbers;

A^* denotes the set of all finite sequences of elements of a set A; empty sequence is included in A^*;

$\emptyset$ denotes the empty set;

$A \cup B$ denotes the union of two sets A, B;

$A \cap B$ denotes the intersection of two sets A, B;

$A \setminus B$ denotes the difference of the sets A and B, i.e. the result is the set of all elements from A that are not present in B;

$\mathcal{P}(A)$ denotes the power-set of a set A, i.e. the set of all subsets of A.

Let A be a finite set. A relation $\phi \subseteq A \times A$ is *anti-symmetric* whenever $((a,b) \notin \phi) \vee ((b,a) \notin \phi)$ holds for all $a, b \in A$ such that $a \neq b$. The relation ϕ is *anti-reflexive* whenever $(a,a) \notin \phi$ holds for all $a \in A$. The relation ϕ is *transitive* if $(a,c) \in \phi$ holds whenever there exists $b \in A$ such that both $(a,b) \in \phi$ and $(b,c) \in \phi$ hold.

A relation $\phi \subseteq A \times A$ is a *strict linear ordering* on A whenever ϕ is anti-symmetric, anti-reflexive and transitive. A set A is *strictly linearly ordered* whenever there exists a strict linear ordering on A.

In this thesis, the term *ordering* always denotes a *strict linear ordering*. Similarly, the term *ordered set* always denotes a *strictly linearly ordered set*.

It is a well-known fact that there exists at most $|A|!$ different (strict linear) orderings of a finite set A.

2.1 Linguistic terms

This section introduces basic "linguistic" terms used throughout the whole thesis. All the terms are defined "in words" as well as in a formal way (as algebraic

structures). It is assumed that the terms *natural language* and its *alphabet* are widely known.

word or *word form* or simply *form* in a text is a sequence of letters from the alphabet of a given natural language, e.g. `here` in English or `dělají` in Czech. In particular, punctuation (comma, dot, semi-colon,...) or a number is also marked up as *form*[1]. Formally, it is expected that for every language there exists a set of all of its words. Sometimes in this thesis, *word* also denotes the morphological analysis of the word form (see below).

morphological category or simply *category* denotes one of the categories that are relevant for the particular language (e.g. `part-of-speech` (PoS), `number`, `gender`, `case`, `person`,...). This is very language-dependent, please refer to [19, 20, 22] for the list of categories used in this thesis.

Each category is identified with the domain of its possible *values*, e.g. {`masculine,feminine,neuter`} for `gender`. A special value `n/a` (non-applicable) is introduced for cases when the category is irrelevant for a particular word form (e.g. `case` for verbs will be set to `n/a`). Categories are not further formalized since they are encoded in *tags*.

tag is known as a combination of values of all categories, i.e. a selection of one value for each category — an example can be found in Section 1.1.

tagset for a natural language: it often happens that some tags (as a combination of category values) are impossible (e.g. verbal tags with dative). The set of all possible tags for the natural language is called *tagset*. Furthermore, *tag* denotes always only a valid combination of category values, i.e. *tag* is always a member of the *tagset*. The tagset is formally just a set of elements (tags or their indexes) usually referred to as T.

lemma — the morphological analysis of a word form, in fact, returns always a set of pairs (`lemma,tag`), where *lemma* is a *base form* of a given word (e.g. nominative singular for nouns). Formally, it is natural to assume that (similarly to word forms) there is a set containing all lemmas of the language, usually referred to as L. Throughout this thesis, *lemma* is mostly considered to be a part of a tag.

morphological analysis of a word w is a subset of $L \times T$. Usually a word form w is replaced directly by its morphological analysis, i.e. the analysis of a sequence of words can be viewed as a sequence of subsets of $L \times T$. With the lemmas "inside" the tags, morphological analysis of a word becomes just a subset of T. The analysis of a sentence is then often referred to as a sequence of "columns" (sets) of tags.

[1] In fact, the definition of what really is a *form* depends on the agreement between the segmentation of a corpus into "tokens" and the morphological analyzer. However, this problem appears on too low a level of processing to be further discussed here.

morphological analyzer is a process that generates morphological analysis for its input (a single word or a text).

morphological generator reconstructs a word form (if possible) from a pair `(lemma,tag)`.

ambiguity of a word is the number of distinct tags assigned by morphological analysis. The aim of this thesis is to perform partial disambiguation by a stepwise elimination of tags that are irrelevant with respect to the sentential context. The *ambiguity* means the *actual* number of `(lemma,tag)` pairs remaining for the word during the process of disambiguation. The term *ambiguity* can be extended to *sentences* in two ways — either as the sum of the ambiguity of the single words or as the sum of surviving readings of the sentence, see below.

disambiguation is either a process or a result of the reduction of ambiguity of the input (word, sentence, text...). A disambiguation can be either *full* (exactly one tag remains for each word; this is what all the stochastic taggers do) or *partial* (more than one possibility is allowed). Extremes of partial disambiguation are no disambiguation (original input is returned) and full disambiguation. Note that the result of a disambiguation is again (formally, see below) a sentence.

sentence is understood as a sentence of a natural language, i.e. as a finite sequence of words. However, most conclusions that use the term *sentence* can be extrapolated without any change also to larger pieces of text. It is always expected that the sentence is already morphologically analyzed. A sentence s of length n can appear in one of the four formal representations: as a sequence of word forms $s=\{w_i\}_{i=1,\ldots,n}$ where $w_i \in W$ for all i from 1 to n, where W is the set of all words of a language (rarely used), in the columns-notation, in the readings-notation or in the dispersed-notation (see immediately below). Throughout this thesis, it is assumed that every sentence contains at least one word.

columns-notation is one of possible representations of a sentence. In this notation, a sentence s of length n is represented as a sequence of the results of morphological analysis of its particular words, i.e. $s = \{h_i\}_{i=1,\ldots,n}$ where $h_i \subseteq L \times T$ (or $h_i \subseteq T$) for all i from 1 to n.

readings-notation is one of possible representations of a sentence. In this notation, a sentence s of length n is represented as a set of all of its readings (see below), i.e. $s \subseteq (L \times T)^n$ or $s \subseteq T^n$.

dispersed-notation is another representation of a sentence. In dispersed-notation, a sentence s is a subset of $T \times \mathbb{N}$, where $(t, i) \in s$ holds if and only if a tag t is present in s on position i. The expressive power of the dispersed notation is obviously equivalent to the power of columns-notation and every sentence can be simply translated from one notation to another.

position i in a sentence usually denotes i-th (running) word of the sentence, i.e. either w_i from the definition of the sentence or h_i from the definition of the columns-notation.

reading of a sentence $s \subseteq (L \times T)^n$ of length n is a selection of exactly one `(lemma,tag)` pair for each position, i.e. a reading r is simply a member of s from the readings-notation. It should be pointed out that under disambiguation of a sentence s also a selection of readings can be understood. Hence, if $s \subseteq (L \times T)^n$ then every (partial) disambiguation d is simply $d \subseteq s$. If $s = \{h_i\}_{i=1,\ldots,n}$ where $h_i \subseteq (L \times T)$, the disambiguation is $d = \{h'_i\}_{i=1,\ldots,n}$ where $h'_i \subseteq h_i$ for all i from 1 to n. It should be always clear from the context which definition is actually used. Note that for an unambiguous sentence (i.e. a sentence containing only one reading) and for any single reading both notations represent the same — a simple sequence of tags. So the definition $r = \{t_i\}_{i=1,\ldots,n}$ where $t_i \in (L \times T)$ for all i from 1 to n is used widely for readings to avoid confusing brackets.

manual reading (in any notation) refers to a manually annotated reading. Sometimes it seems useful to make it possible for annotators to select more than one reading (e.g. whenever the annotator is not sure) — but this is not the case in this thesis.

Since the operator $|s|$ can be ambiguous when applied to a sentence s, the following useful operators are defined: `length`(s) denotes the length of sentence in positions (i.e. the number n in the definitions above), `ambiguity`$(s) = \sum_{i=1}^{n} |h_i|$ denotes the ambiguity of sentence $s = \{h_i\}_{i=1,\ldots,n}$ and `readings`$(s) = |s|$ denotes the ambiguity of sentence s in the readings-notation (when $s \subseteq (L \times T)^n$).

2.2 Formal languages

In the thesis, it is assumed that the reader is familiar with the hierarchy of languages introduced by Noam Chomsky ([38]) and with the corresponding classes of acceptors — *finite-state machines* (FSM), *pushdown automata* (PDA) and *Turing machines* (TM). These basic terms are briefly introduced. Formal definitions and other details can be found in a lot of publications, e.g. in [3].

Non-deterministic finite-state machine is a 5-tuple $\mathcal{A} = (Q, X, \delta, I, F)$ where Q is the set of states, X is the input alphabet, $\delta : Q \times X \times Q$ is the transition relation, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of final (accepting) states. For each input sequence $s \in X^*$, the machine $\mathcal{A}$ starts in an initial state, uses transition relation δ to "eat up" the elements of s and ends in some state $q \in Q$ when the whole s is processed.

A *computation* of a FSM $\mathcal{A}$ for a sequence $s = x_1, \ldots, x_n$ of length n (for $x_i \in X$ for all i from 1 to n) is a sequence $c = q_0, \ldots, q_n$ (for q_i in Q for all i from 0 to n) such that $q_0 \in I$ and $(q_{i-1}, x_i, q_i) \in \delta$ for all i from 1 to n. The computation is *accepting* whenever $q_n \in F$.

If there exists such an accepting computation of $\mathcal{A}$ on s such then $\mathcal{A}$ *accepts* the input s. If there is no accepting computation of $\mathcal{A}$ on s then the sequence s is *refused* by $\mathcal{A}$.

For a machine $\mathcal{A}$, let $L(\mathcal{A}) \subseteq X^*$ denote the set of all sequences accepted by $\mathcal{A}$. A language $L \subseteq X^*$ is called *regular* if and only if there exists a FSM $\mathcal{A}$ such that $L = L(\mathcal{A})$.

Usually it is allowed to "under-define" the transition relation δ, i.e. for some $(q, x) \in Q \times X$ there exists no q' such that $(q, x, q') \in \delta$. In such a situation, if a machine in state q reads x from the input, the input is not accepted.

The machine $\mathcal{A}$ is called *deterministic* whenever the following two conditions hold simultaneously: $|I| = 1$ and for each $q \in Q$ and $x \in X$ there exists at most one $q' \in Q$ such that $(q, x, q') \in \delta$.

It is well known (see [3]) that deterministic FSMs accept exactly regular languages, i.e. any non-deterministic FSM can be determinized. The determinization, however, sometimes causes exponential growth of the number of states. In other words, the determinization cannot be generally accomplished in a polynomial time.

It is also well known that the class of regular languages is closed for the operations of intersection, union and difference, i.e. for two FSMs $\mathcal{A}$, $\mathcal{B}$ a FSM $\mathcal{C}$ can always be constructed such that $L(\mathcal{C}) = L(\mathcal{A}) \cap L(\mathcal{B})$, $L(\mathcal{C}) = L(\mathcal{A}) \cup L(\mathcal{B})$, $L(\mathcal{C}) = L(\mathcal{A}) \setminus L(\mathcal{B})$, respectively. Moreover, if the input machines $\mathcal{A}$, $\mathcal{B}$ are deterministic, the result $\mathcal{C}$ is also deterministic. In all three cases, the construction of the machine $\mathcal{C}$ consumes polynomial time (by the size of the input machines $\mathcal{A}$ and $\mathcal{B}$).

Regular languages can also be defined by so-called *regular expressions* — they are equivalent to FSMs and the formal definition can be found in [3].

In natural language processing, it is also useful to define finite-state *transducers* (FST, see [29]). Finite-state transducer A is a 6-tuple $(Q, \Sigma, \Delta, \delta, I, F)$, where Q is the set of states, Σ is the input alphabet, Δ is the output alphabet, $\delta : Q \times \Sigma \times Q \times \Delta^*$ is both transition and output relation, $I \subseteq Q$ is the set of initial states and $F \subseteq Q$ is the set of accepting states. An input word $u \in \Sigma^*$ is accepted by A with the output $w \in \Delta^*$ whenever there exists a computation of A for u such that it finishes in a state from F and the concatenation of all output strings of individual transitions in the computation is equal to w.

A *sequential* finite-state transducer $A = (Q, \Sigma, \Delta, \delta, I, F)$ is a FST with the following restrictions: $|I| \leq 1$ and δ must be unambiguous, i.e. for each pair $(q, \sigma) \in Q \times \Sigma$ there exists at most one pair $(q', w) \in Q \times \Delta^*$ such that $(q, \sigma, q', w) \in \delta$.

Pushdown automata are not formally used in this work, but they are sometimes referred to. Informally, a pushdown automaton is a FSM that has an infinite stack at its disposal. It can store any information on the stack and pop it from the stack in the reverse order. Formal languages accepted by PDAs are called *context-free* languages.

Non-deterministic *Turing machine* is a 6-tuple $\mathcal{T} = (Q, X, \Sigma, \delta, I, F)$, where Q is the set of states, X is an input alphabet, Σ is a working alphabet containing a special symbol λ (empty symbol), δ is the transition relation, I is the set of initial states and F is the set of accepting states. $\mathcal{T}$ has one input tape

(read-only) with one head and one working tape (read-write) with one head. It starts its computation with an input word from X^* written on the input tape and with the rest of the input tape being set to λ. Then it continues (non-deterministically) using the transition relation $\delta \subseteq Q \times X \times \Sigma \times Q \times M \times \Sigma \times M$, where $M = \{\leftarrow, \cdot, \rightarrow\}$ denotes the set of possible moves of the head on the tape (move left, stay, move right). Each $(q, x, \sigma, q', m, \sigma', m') \in \delta$ specifies that if the machine is in state q, one head reads x on the input tape and the other head reads σ on the working tape then the machine moves m on the input tape, it writes σ' and moves m' on the working tape and switches to state q'. The input word is accepted whenever there exists a computation such that the machine enters a state from F. The language defined by a particular TM is the set of all words accepted by the TM. The set of all languages accepted by all TMs is the class of *general languages*.

A Turing machine M is *deterministic* whenever, roughly speaking, the relation δ is a partial function. As deterministic TMs accept exactly general languages (i.e. they are equivalent to non-deterministic TMs), any formal definition is neither given, nor used here.

A Turing machine M is *linear* whenever it uses linear space for its operation (with respect to the length of the input). Formally, for every linear Turing machine there exists an integer k such that for any input of length n there is at most kn positions used on the working tape. Technically, a linear Turing machine M contains two special symbols $\tau, \chi \in \Sigma$. When the computation of M starts on an input word w, M allocates a part of the working tape of the size $k|w|$ by enclosing it by τ and χ. Whenever M reads τ or χ from the working tape, it must keep the symbol on the tape and move right or left, respectively. Whenever M reads a symbol different from both τ and χ, it is not allowed to write neither τ nor χ. Linear Turing machines accept the class of *context languages*.

A Turing machine M *decides* a language L (over the alphabet X) whenever M stops for each $\alpha \in X^*$ and M accepts exactly the language L. The class of *recursive* languages is the class of all languages L such that there exists a Turing machine that decides L. It is well-known that the class of context languages is a subclass of the class of recursive languages (see [3]).

Non-deterministic *Turing transducer* is a 7-tuple $T = (Q, X, Y, \Sigma, \delta, I, F)$. Its operation is identical to an *underlying* Turing machine $(Q, X, \Sigma, \delta, I, F)$ with the only exception: it contains also one output tape (write-only) and the transition function $\delta \subseteq Q \times X \times \Sigma \times Q \times M \times \Sigma \times M \times Y^*$ defines (in its last component) the output. An input string s is accepted by T whenever the underlying machine accepts s. A Turing transducer T is *deterministic* whenever $|I| \leq 1$ and the transition relation δ is a function, i.e. for each $(q, x, \sigma) \in Q \times X \times \Sigma$ there exists at most one 5-tuple $(q', m, \sigma', m', \alpha) \in Q \times M \times \Sigma \times M \times Y^*$ such that $(q, x, \sigma, q', m, \sigma', m', \alpha) \in \delta$.

Classes of languages accepted by FSMs, PDAs, linear TMs and TMs can be also generated by formal grammars. Regular languages are generated by *regular grammars*, context-free languages are generated by *context-free grammars*, context languages are generated by *context grammars* and general languages are generated by *general grammars*.

General grammar is a quadruple $\mathcal{G} = (N, T, P, S)$, where N is a set of non-terminals, T is a set of terminals, P is a set of rewriting rules and $S \in N$ is the starting symbol. P is a set of rules of the form $\alpha \to \beta$, where $\alpha, \beta \in (N \cup T)^*$ and $(\exists x \in N)(\exists \alpha', \alpha'' \in (N \cup T)^*)(\alpha = \alpha' x \alpha'')$.

Derivation in the grammar $\mathcal{G}$ is a sequence

$$S \to \alpha_1 \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \tag{2.1}$$

such that there exist strings $\beta_i, \beta_i', \beta_i'', \beta_{i+1}, \beta_{i+1}', \beta_{i+1}'' \in (N \cup T)^*$ such that $\alpha_i = \beta_i' \beta_i \beta_i''$ and $\alpha_{i+1} = \beta_{i+1}' \beta_{i+1} \beta_{i+1}''$ and $\beta_i \to \beta_{i+1} \in P$ for all i from 1 to $n-1$. Let $w \in (N \cup T)^*$. If there exists a sequence of words $\alpha_1, \ldots \alpha_n$ such that $\alpha_n = w$ and (2.1) is a derivation in the grammar $\mathcal{G}$, then the word w can be *generated* by $\mathcal{G}$, shortly $S \to_{\mathcal{G}}^* w$.

The language generated by the grammar $\mathcal{G}$ is $L(\mathcal{G}) = \{w; w \in T^* \wedge S \to_{\mathcal{G}}^* w\}$.

Regular, context-free and context grammars have restrictions in the sets of rules. A context grammar can contain only rules of the form $\alpha X \beta \to \alpha \gamma \beta$, where $X \in N$, $\alpha, \beta, \gamma \in (N \cup T)^*$. Context-free grammar can contain only the rules of the form $X \to \alpha$, where $X \in N$ and $\alpha \in (N \cup T)^*$. Regular grammars can contain only the rules of the form $X \to \beta Y$ or $X \to \beta$, where $\beta \in T^*$ and $X, Y \in N$.

2.3 NP-completeness

The primary objective of the thesis is not to investigate NP-completeness of presented problems, but since the terms from the theory of NP-completeness are used, basic definitions are given here (see [32]).

A formal language L (over an alphabet X) belongs to the *class NP* (shortly $L \in NP$) if and only if the problem $(x \in L)$ can be decided by a non-deterministic Turing machine T in polynomial time (by the size of x) for every $x \in X^*$ — if there is at least one accepting computation of T, then $x \in L$, otherwise $x \notin L$.

A language L (over an alphabet X) is *polynomially-Turing-reducible* to a language M (over Y) if the problem $(x \in L)$ can be decided by a deterministic TM in polynomial time (by the size of x) with oracle M[2] for all $x \in X^*$.

A language L is *NP-hard* whenever every language $M \in$ NP is polynomially-Turing-reducible to L. L is *NP-complete* whenever it is NP-hard and $L \in$ NP. If M is a NP-complete language polynomially-Turing-reducible to a language L, then L is NP-hard.

2.4 Abstract rules

First of all, formal definition of "rules" employed in this work is given in Section 5.1. Currently only "abstract" term *disambiguation rule* is introduced to simplify the formalism in the following chapters.

[2]The oracle M is expected to decide the problem $(y \in M)$ for any $y \in Y^*$. During the computation the oracle can be asked any number of times and the time consumed by the oracle itself is not counted into the total time consumed by the deterministic TM in question.

disambiguation rule is an algorithm M such that M reads a sentence and outputs its partial disambiguation (w.r.t. the grammar knowledge encoded in the machine M). The algorithm M must always stop.

Note: In this "virtual" definition of a disambiguation rule, it is not essential whether the rule operates on columns-notation or on readings-notation.

For a sentence s and a disambiguation rule M, $M(s)$ denotes the fact that the algorithm M *accepts* the sentence s, while $\neg M(s)$ denotes the fact that s is *refused* by M.

The *set of disambiguation rules* $\mathcal{M}$ is defined as a serial composition of participating rules. The order of rules in the composition can be significant, see Chapter 5. For a set of rules $\mathcal{M}$, $\mathcal{M}(s)$ holds for any input s such that $M(s)$ holds for all the rules $M \in \mathcal{M}$. Otherwise $\neg\mathcal{M}(s)$ holds. A sentence s is $\mathcal{M}$-*correct* whenever $\mathcal{M}(s)$ holds. Otherwise, the sentence s is $\mathcal{M}$-*incorrect*.

Discussion: For a set of disambiguation rules M, there are generally two ways of handling the input sentences that are "incorrect" w.r.t. the rules. In the first approach, the machine M accepts only M-correct sentences. M-incorrect sentences are not accepted by M and the output of M is not significant in this case (no surprise so far). In the second approach, M accepts all inputs (even M-incorrect) and it returns empty output (i.e. either the sentence with empty tag columns, or empty set of readings depending on the particular notation used by M) for M-incorrect inputs. Since the task of checking the "grammaticality" of the input can be separated from the disambiguation, the first approach is generally used — M can refuse some (non-grammatical) inputs and the output is returned only for accepted inputs. In fact, accepting all inputs and determining non-grammaticality by null output is a more difficult task for simpler machines (e.g. FSMs).

In this context, it is useful to realize that not all disambiguation rules must essentially recognize incorrect sentences. In particular, there exist disambiguation rules that never refuse unambiguous sentences but they are able to delete tags from ambiguous inputs. For such a rule M the formula $M(s)$ holds for any sentence s. This phenomenon is investigated later in Chapter 5.

Chapter 3

Issues of Competent Annotation

3.1 Errors of stochastic taggers

For a long time the task of morphological disambiguation (at least of Czech) has been reserved exclusively for stochastic taggers (and manual annotation, of course). As well as any other model of anything with the complexity of a natural language, stochastic taggers are never 100% successful in annotating language corpora. Linguists that used stochastically tagged corpora pointed out that a lot of errors could be avoided by applying a relatively simple linguistic knowledge.

For example, it is known that every vocalized preposition can stand only in front of well defined set of word forms. Usually, this set of words can be described by a union of regular expressions. Four years ago a detailed study of disambiguation of the Czech word *se* has been elaborated, which is ambiguous between the preposition and the reflexive pronoun/particle. The study has shown that (at that time) the stochastic taggers committed about 40% errors in tagging the word *se*. A lot of these errors could be removed by applying grammatical knowledge, see [16]. On the other hand, at present it seems that stochastic taggers considerably improved — they commit only about 2% errors with *se* — explanation of this improvement is not further discussed here.

Another simple observation — a preposition cannot stand immediately in front of a verb (in any well formed Czech sentence) — leads to concrete figures: more than 12,000 occurrences of a preposition immediately followed by a verb can be found in the stochastically annotated corpus SYN2000. Of course, it is only 0.012% of the whole corpus. But these errors can be removed very easily — and the quality of the corpus will thus be improved.

With the increasing use of tagged corpora the demands for its quality became more frequent. Imagine, for example, that a linguist would like to look for a phrasal use of a preposition immediately followed by a verb. Although the phenomenon is very rare in Czech, the linguist will be flooded with thousands of concordance lines.

The following subsections try to summarize the motivations and goals that

led to the idea of formalizing a set of rules that describe the grammatical system of Czech and to the development of a mechanism that implements these rules.

3.2 Quality of corpora

It can be objected that problems of a single tagger could be solved, for example, by the voting of several different taggers. However, the examples in the previous section show that simple knowledge of grammar can perform well with minimal costs (and with no training data).

This is a big step toward considerations about the quality of corpora tagging, which has been rarely discussed so far. Of course, it is necessary to have large corpora (i.e. to run for quantity). Linguistically competent rules allow us also to say something about the *quality* of a corpus, both about the grammaticality of raw texts stored in the corpus and about the quality of the tagging of texts.

A practical advantage claimed by stochastic taggers (and also by some rule-based systems with probabilistic rules) is that they can annotate any input. In particular, they can annotate also a piece of text that is ungrammatical. This is exactly the feature that makes the taggers unable to detect incorrect sentences. It can be objected that stochastic taggers can also somehow mark up non-grammatical input, e.g. by setting a limit of sentence probability that represents a "correct" sentence. However, since corpora containing annotation of errors are not available yet, it could be difficult to set up this limit.

On the other hand, human-written grammar can identify sentences that do not correspond to the grammar. Moreover, it can be precisely stated which types of errors can be detected by the rules.

Goal 3.1 *Be able to check fully disambiguated input.*

From this goal it follows that the grammar can be employed in checking both manual annotation and automatically annotated corpora. It also opens a possibility of marking up grammatically incorrect sentences in corpora.

Generally, any sentence (as a sequence of tags) can be refused by the grammar because of one of the following reasons:

- ungrammaticality of the raw text,
- error of manual annotation,
- both sentence and its annotation are correct but there is an incorrect rule in the grammar.

The great difference between stochastic taggers and human-written rules is that the operation of rules can always be traced and an error in the rules can be found and *corrected*. In a simpler case the competent linguist who wrote the rule can correct it.

There could also appear more complex problems, such as errors in morphological analysis, problems in the tagset design, or more intricate phenomena of natural language in question.

In all cases, after processing a corpus with a grammar at hand, it is ensured that the language phenomena described by the grammar are correctly tagged in the annotated corpus.

Of course, the phenomena outside the grammar remain unchecked. For example, if the grammar doesn't account for predicate-subject agreement then potential errors in this agreement cannot be detected.

It could be objected that stochastic taggers can also be used to check manual annotation — just annotate automatically the same text and look for differences. However, so far stochastic taggers are much less reliable than human annotators. In particular, the majority of the differences found are errors of the stochastic tagger itself.

3.3 Conservative and total grammars

Goal 3.2 *The grammar must not claim sentences to be grammatically incorrect if they are grammatical in the language.*

The Goal 3.2 seems natural, however, it is good to state it explicitly. It says, roughly speaking, that the grammar must not produce errors in correct input sentences.

In some approaches, it is useful to set an opposite goal. Mainly, if the grammar has to achieve full disambiguation then the rules could be ordered from "safe" to "dangerous". Safe rules can be understood as those fitting the Goal 3.2. Dangerous rules, on the other hand, can depart from an uncertain hypothesis to obtain less ambiguity with some level of uncertainty.

Let any grammar that fits Goal 3.2 be denoted a *conservative* grammar. Since "correct in the language" is a vague term, formal definition of conservative grammar must be very abstract:

Definition 3.3 (Conservative grammar) *Let $\mathcal{L}$ be the set of all sentences of a particular natural language. Then a grammar $\mathcal{G}$ is* conservative *(w.r.t $\mathcal{L}$) if and only if $(\forall w)(w \in \mathcal{L} \Rightarrow w \in L(\mathcal{G}))$ holds.*

In other words, here $\mathcal{L}$ represents the performance of a particular natural language. Hence checking whether a (non-trivial) grammar is conservative with respect to $\mathcal{L}$ is an unrealistic task.

The opposite term to *conservative* grammar is the term of *total* grammar. A total grammar ensures that any sentence incorrect in the language will be refused by the grammar. In other words, any total grammar generates a subset of a natural language.

Definition 3.4 (Total grammar) *Let $\mathcal{L}$ be the set of all sentences of a particular natural language. Then a grammar $\mathcal{G}$ is* total *(w.r.t $\mathcal{L}$) if and only if $(\forall w)(w \in L(\mathcal{G}) \Rightarrow w \in \mathcal{L})$ holds.*

The initial objectives of this thesis are

- correct disambiguation of any morphologically analyzed (Czech) corpus and

- verification of a raw text grammaticality and verification of an existing disambiguation of a (Czech) corpus, see Goal 3.1.

The verification of corpora correctness (in all applications — checking taggers, manual annotation or raw text grammaticality) by a conservative grammar is not a real problem. The worst thing that can happen is that an incorrect sentence will be treated by the grammar as correct — which perfectly fits Goal 3.2.

However, a total grammar can treat a lot of sentences as incorrect, while they are in fact correct in the language. In most applications[1] this behavior is confusing — the importance of Goal 3.2 is clear.

On the other hand, the total and conservative grammar is required to perform the full correct disambiguation of a corpus. It is (very probably) impossible to write down total and conservative grammar generating exactly the whole language, mainly because of expected high cardinality of the grammar (measured by the number of rules)[2].

A conservative grammar produces generally only partial disambiguation of a corpus. The ambiguity that remains in the corpus after the application of the grammar must be removed by separate modules (whenever the full disambiguation is required) — by a stochastic module or by a total grammar.

3.4 Output of partial disambiguation

Let T denote the tagset[3] and let S be a sentence of length n to be disambiguated. The sentence S has been morphologically analyzed, i.e. each word of the sentence has been assigned the set of its possible tags. Let w_i (for i from 1 to n) denote these tag sets.

Let's highlight that there are two possible points of view that can be used in looking at an ambiguous sentence (see Section 2.1). A sentence S of length n can be seen as a sequence of tag sets (columns-notation), but it is also possible to look at it as a set of all possible readings (readings-notation). In the columns-notation, the equation $S = \{h_i\}_{i=1}^{n}$ holds (where $h_i \subseteq T$ for all i from 1 to n), whereas in the readings-notation $S \subseteq T^n$ holds.

There is no difference between the two notations either directly after morphological analysis (i.e. in the input), or in the full disambiguation. However, the selection of a notation plays the key role in the task of partial disambiguation.

The difference between the two notations can be demonstrated by the following simple Czech example. Let *Věž chrání most.*[4] be the sample sentence. It is assigned the following morphological analysis:

[1] Including the manual correction of stochastic errors found by the rules or a grammar checker that should be used in a text editor.

[2] And also because there is a lot of sentences such that their grammaticality is still an object of linguistic discussion.

[3] Let lemmas be a part of tags for the time being to simplify mathematical formulae.

[4] In English, *The tower defends the bridge.* or *The tower is defended by the bridge.*

Věž	chrání	most	.
noun-fem-sg-nom	verb	noun-masc-sg-nom	punct
noun-fem-sg-acc		noun-masc-sg-acc	

From the Czech grammar it is known that the verb *chrání* can never have either two nominatives or two accusatives in its valency frame. This fact enables us to perform some disambiguation of the input sentence. Namely, only the following two readings remain in the sentence:

Věž	chrání	most	.
noun-fem-sg-nom	verb	noun-masc-sg-acc	punct
noun-fem-sg-acc	verb	noun-masc-sg-nom	punct

The other readings are forbidden by grammatical rules. In the readings-notation of the sentence it will be taken into account. However, if the disambiguated sentence is stored in the columns-notation, we shall end up with the original sentence, since no single tag can be deleted.

This happens since the deletion of tags in one position is dependent on the situation in another position — and this is the feature that cannot be described by the columns-notation.

So it seems quite clear that the readings-notation should always be used. From the theoretical point of view, there is nothing to object. However, in practice it can simply happen (for long and largely ambiguous sentences) that the number of readings cannot be handled with existing computational power[5].

3.5 Shallow syntactic analysis

To achieve the rule-based morphological disambiguation, it is obviously necessary to employ all "higher" levels of natural language — syntactic, semantic and the knowledge of the world. However, if partial disambiguation is enough, a lot of work can be done by applying simple syntactic rules.

Moreover, it is not necessary to build up any structure (usually a tree) upon the sentence. Most phenomena can be described on *flat* sentence instead. Here *flat* is understood as a "sequence of word forms" (including their morphological analysis). Hence the rules in fact have to distinguish between correct and incorrect sequences of word forms (with their morphological analysis).

Goal 3.5 *Every rule must be defined over a flat sequence of words.*

This is, of course, another vague goal — mainly because any hierarchical relations over a sentence can be encoded in the data structures of the programming language interpreting the rules.

On the other hand, the idea behind this is simply to avoid explicit generation of any kind of syntactic tree. The implementation should still be able to handle some relations between words in the sentence (but these relations need not necessarily be dependency relations, e.g. a coordination). This is usually called

[5]Consider a sentence of 30 words with ambiguity approx. 2 (which is *very* low) — then you end up with 2^{30} readings...

shallow or *surface* grammar — hence the title *surface grammatical rules* (SGR) is sometimes used for the disambiguation rules in this thesis.

The surface grammatical rules usually work as follows: the rule first defines a *configuration* (sequence of positions) that must be present in the sentence to match the rule. When a part of an input sentence matches the configuration, *executive* part of the rule appears on the stage — it specifies what has to be done (e.g. the deletion of tags).

Sometimes it can happen that a configuration of a rule matches more sub-sequences of a sentence. The linguists naturally want to apply the rule to all possible sub-sequences of the sentence, hence it is necessary to introduce "non-deterministic" searching for the matching sub-sequences.

Good practical question is what expressive power of the rules is enough to cover acceptable large piece of the (Czech) grammar. When the work on this thesis has started, no investigation or hints were available on this topic. So, it was necessary to adopt the following goal:

Goal 3.6 *Allow the existence of any algorithmic rule.*

Chapter 4

Negative n-grams

This chapter introduces *negative n-grams* that are able to distinguish grammatically correct and incorrect readings of a sentence. It formally defines negative n-grams, investigates the possibilities of their automatic retrieval from tagged corpora and the ways of the application of negative n-grams to the corpora and ends up with the considerations that lead the author to develop a more complex formalism to implement grammatical rules.

4.1 Definition of negative n-grams

Negative n-grams are simple rules that determine whether a reading of a sentence is correct (i.e. grammatical). They have been described in [4] in a detailed way, however, a short description is included here for a better understanding of surface rules.

An n-gram usually denotes a sequence of n tags. *Negative* n-grams emerged from the observation that some tag sequences are inadmissible in any well-formed sentence of the particular natural language.

For example, in Czech a finite verb[1] cannot be immediately followed by another finite verb — hence two finite verbs form a *negative bigram*[2].

Further investigation of this observation leads us to extending negative n-grams with context — for example, two verbs are still impossible if there is an arbitrary number of adverbs between them. Generally, a negative n-gram is defined by the formula

$$t_1 C_1 t_2 \ldots t_{n-1} C_{n-1} t_n \tag{4.1}$$

where t_i are tags (for i from 1 to n) and C_j are sets of tags (for j from 1 to $n-1$). The tags t_i are the tags from the original n-gram and C_j are contexts that do not violate the n-gram's ungrammaticality. In particular, our bigram of finite verbs can be rewritten as follows:

$$\text{FINITE\_VERB} \quad \{\text{ADVERB}\} \quad \text{FINITE\_VERB}.$$

Of course, we can find many more negative n-grams in a natural language.

[1] I.e. past participle, imperative or present or future verb form.

[2] A simple negative bigram in English is an article followed by a finite verb.

Definition 4.2 *Let T be a tagset. Let $b = t_1 C_1 t_2 \ldots t_{n-1} C_{n-1} t_n$ ($t_i \in T$ for all i from 1 to n and $C_i \subseteq T$ for all i from 1 to $n-1$) be a negative n-gram and let $r = k_1 k_2 \ldots k_m$ ($k_i \in T$ for i from 1 to m) be a sequence of tags (a reading). Then the reading r* matches *the negative n-gram b whenever there exist integers $i_1, \ldots, i_n$ such that the following conditions hold simultaneously:*

$$1 \leq i_1 < i_2 < \ldots < i_n \leq m$$
$$(\forall j)\left((1 \leq j \leq n) \Rightarrow (k_{i_j} = t_j)\right)$$
$$(\forall p)(\forall j)\left((\ 1 \leq p < n \ \wedge \ i_p < j < i_{p+1} \ ) \Rightarrow (k_j \in C_p)\right)$$

Definition 4.3 *Let B be a finite set of negative n-grams and let r be a reading. Then r is B-*correct *whenever r matches no negative n-gram from B.*

*An (ambiguous) sentence S is B-*correct *whenever there exists a reading $r \in S$ such that r is B-correct.*

Let formula $B(r)$ denote that a reading r is B-correct (for a set B of negative n-grams). Similarly, $\neg B(r)$ denotes that r is *not* B-correct (cf. Section 2.4).

It is useful to recall that there exist also negative n-grams for $n > 2$ such that all $(n-1)$-grams that can be made from the n-gram are not negative. In Czech, any sentence containing uninterrupted sequence of three word forms *se* is always incorrect, while a sequence of two word forms *se* can appear in a grammatical sentence. Hence three adjacent occurrences of *se* represent a negative trigram in Czech.

Negative n-grams are a very powerful and simultaneously simple tool for testing hand-made corpora — if a sequence of tags matches a negative n-gram, something goes wrong there. The efficiency of negative n-grams has been demonstrated on the German corpus NEGRA, see [4, 13].

Employing negative n-grams in disambiguation requires further investigation. It has been proved (see [5]) that checking whether an ambiguous sentence S is B-correct (for a set B of negative n-grams) is an NP-hard problem. A particular set of negative n-grams can be pre-compiled into a finite-state machine that reduces the complexity to the polynomial one, see Section 4.5.

4.2 Negative n-grams and unambiguous input

This section investigates the way of processing non-ambiguous input with negative n-grams. It is obvious that every negative n-gram forms a simple regular expression. The union of all such regular expressions is then able to check whether an input unambiguous sentence matches at least one negative n-gram (i.e. the sentence is incorrect w.r.t. the set of negative n-grams at hand).

Let T be a tagset. Let B denote a finite set of negative n-grams of the form (4.1). For each negative k-gram $b = s_1 C_1 s_2 \ldots s_{k-1} C_{k-1} s_k$ ($s_i \in T$ for all i from 1 to k and $C_i \subseteq T$ for all i from 1 to $k-1$), a FSM $\mathcal{A}_b = (Q_b, T, \delta_b, q_b^0, \{q_b^k\})$ is defined such that any input reading r is accepted by $\mathcal{A}_b$ if and only if r matches the negative k-gram b.

Let $Q_b = \{q_b^0, q_b^1, \ldots, q_b^k\}$. The states $q_b^1, \ldots, q_b^{k-1}$ replace, roughly speaking, C_i elements, respectively. The states q_b^0 and q_b^k stand in front of and after the

whole b, respectively. The state q_b^0 is the initial state, q_b^k is the final (accepting) state. Transition relation δ_b is defined as follows:

$$(\forall i \in \mathbb{N})\,((0 \leq i < k) \Rightarrow (q_i, s_{i+1}, q_{i+1}) \in \delta_b)$$
$$(\forall i \in \mathbb{N})\,((0 < i < k) \Rightarrow (\forall t \in C_i)\,((q_i, t, q_i) \in \delta_b))$$

Other transitions remain undefined. It is obvious that $\mathcal{A}_b$ accepts an input reading r if and only if r matches b. Whenever the automata $\mathcal{A}_b$ are generated for all negative n-grams $b \in B$, it is possible to create their union $\mathcal{A}_B$ — this FSM accepts exactly all readings that match at least one negative n-gram from the set B.

The FSM $\mathcal{A}_B$ can then be applied to manually or automatically disambiguated corpus to check whether the annotation is B-correct. Since any FSM runs in linear time (by the size of input), this is a very efficient technique to check the annotation.

4.3 Retrieval of negative n-grams

The set of negative n-grams can be obtained in two ways:

- by employing competent people (i.e. linguists describing a particular natural language) who will create the set of negative n-grams using their knowledge of the system of the given language;
- by inferring the set from the training data.

Using any of the above approaches alone can lead to more or less corrupted sets — linguist's imagination is not absolute and the training data always contain some errors.

One of the effective ways of combining both methods mentioned above has been used to process German corpus NEGRA (see [4]). The algorithm retrieves the negative n-grams from the corpus with huge intervention of the linguists. Let's illustrate the algorithm on bigrams (i.e. pairs of tags).

Let T be a tagset. Let E denote the set of all bigrams that exist in the training corpus. The first iteration of specifying the set B of negative bigrams is defined simply as $B = (T \times T) \setminus E$.

Since it can be assumed that the training corpus is neither representative, nor correct, the set B must be processed manually by the linguists to exclude any bigram such that it is possible in the system of language, but it has not been seen in the corpus. It is also possible that some bigrams that have been seen in the corpus are negative in fact — so it is useful also to process manually all the bigrams with "low" counts from E.

Now the hand-annotated corpus can be checked by the set B (still without contexts) to eliminate at least some possible errors of the annotators.

The context $C_{(a,b)}$ of a particular negative bigram $(a, b) \in B$ can be also achieved from the corpus. It is a complement (of the whole tagset T) of the set $\bar{C}_{(a,b)}$ that can be retrieved as follows.

Start with all "positive" trigrams (a, x, b), i.e. those found inside a sentence in the corpus — put all such tags x into $\bar{C}_{(a,b)}$. Then proceed with tetragrams (a, z, y, b) such that $z \notin \bar{C}_{(a,b)}$ and $y \notin \bar{C}_{(a,b)}$, continue with pentagrams and so on until the maximum sentence length in the corpus is reached. Then let $C_{(a,b)} = T \setminus \bar{C}_{(a,b)}$.

Once the sets $C_{(a,b)}$ are retrieved for all bigrams $(a, b) \in B$, they should be processed manually by linguists to eliminate incorrectness. Then the hand-annotated corpus can be checked for errors and the next iteration of the sets $C_{(a,b)}$ can be retrieved. The process can continue in cycle until linguists are tired or the corpus is completely correct.

The results of this approach have also been published in the paper [4]. However, the applicability of the algorithm depends largely on the size of the tagset — in the case of German, the tagset contains less than 60 tags. In the case of Czech, the tagset contains more than 4300 tags, which brings about some problems for a manual processing of the set B of maximal size about 4300^2.

Similar problem emerges if negative n-grams are to be retrieved for $n > 2$.

4.4 Negative n-grams and ambiguous input

Negative n-grams have been introduced and practically used to process a manually annotated corpus. But is it possible to disambiguate with negative n-grams? Several attempts of converting negative n-grams into a disambiguation machine have been made, however, real effective application has been proposed by Jan Hajič — it is described in Section 4.5. The current section introduces generic (non-effective) algorithm that performs the disambiguation and defines the output of disambiguation with negative n-grams. Then the section summarizes previous attempts in searching the effective disambiguation algorithm, since it is a good practice for understanding negative n-grams.

Definition 4.4 *Let T be a tagset and let B be a finite set of disambiguation rules. Let $S \subseteq T^n$ be an input sentence of length n and $S' \subseteq S$ its partial disambiguation. Then S' is the* B-conservative *partial disambiguation of S whenever $(\forall r)(r \in S \wedge B(r) \Rightarrow r \in S')$ holds. S' is the* B-total *partial disambiguation of S whenever $(\forall r)(r \in S \wedge \neg B(r) \Rightarrow r \notin S')$ holds. The partial disambiguation S' is the* B-total-conservative *disambiguation of S whenever S' is both B-total and B-conservative.*

B-total-conservative disambiguation S' in this definition can be acquired from the input sentence S using a so-called *generic disambiguation algorithm* defined in the paper [5]. In fact, generic disambiguation algorithm processes the set S and checks each particular reading whether it is B-correct. All B-correct readings are put into S'. The algorithm is obviously exponential in time (by the size of the input sentence), but it also ensures that there exists exactly one B-total-conservative disambiguation for every sentence[3].

[3]The marginal problem seems to emerge for sentences that are ungrammatical w.r.t. the set B — all readings are B-incorrect in such a sentence. As "empty" disambiguation is formally

B-total-conservative disambiguation represents the result that is naturally expected of any disambiguation algorithm based on negative n-grams B — it removes exactly all readings that violate the knowledge of the language system encoded in B. In the case of columns-notation, it becomes a little bit less trivial:

Definition 4.5 *Let T be a tagset and let B be a finite set of negative n-grams. Let $S = \{h_i\}_{i=1}^{n}$ be the sentence of length n with $h_i \subseteq T$ for all i from 1 to n and $S' = \{h'_i\}_{i=1}^{n}$ be a partial disambiguation of S where $h'_i \subseteq h_i$ for all i from 1 to n. Then S' is*

- B-conservative *disambiguation of S whenever for each j from 1 to n and each $t \in h_j$ the following implication holds:*

$$[((\exists m_1 \in h_1) \ldots (\exists m_n \in h_n))(m_j = t \wedge B(m_1 \ldots m_n))] \Rightarrow (t \in h'_j).$$

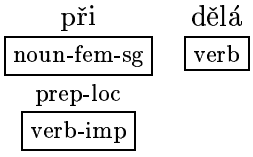
- B-total *disambiguation of S, whenever for each j from 1 to n and each $t \in h'_j$ there exist tags $k_1 \in h'_1, \ldots, k_{j-1} \in h'_{j-1}, k_{j+1} \in h'_{j+1}, \ldots k_n \in h'_n$ such that $B(k_1 \ldots k_{j-1} t k_{j+1} \ldots k_n)$ holds.*

S' is a B-total-conservative disambiguation of S whenever it is both B-total and B-conservative.

At the time the work on negative n-grams started, a simple assumption was made that it would not be possible to use readings-notation because of its exponential time and space consumption.

In this situation, at first we have been looking for a machine that would use columns-notations. Only later it was proved for the FSMs that in practice the number of really different readings is not so high and that it is manageable, see Section 4.5.

We have tried to design an effective machine (a finite-state transducer, in the particular case) that could employ negative n-grams in the disambiguation task. The simplest strategy has been to look at each negative n-gram through *fixpoint–disambiguationpoint* (FD) glasses. It can be simply shown on a bigram PREP VERB. With FD glasses, this negative bigram defines two *disambiguation rules*. The first rule fixes the preposition on the first position and says that any verbal tag in the subsequent position must be deleted. The second rule fixes the verb on the second position and says that any prepositional tag from the previous position must be deleted. This can be illustrated on the sequence *při dělá*:



Since *dělá* is a safe verb, the second rule can be used to delete prepositional tag from *při*. Tags that remain are highlighted by frames.

(by definition) a disambiguation, ungrammatical sentences also have their B-total-conservative disambiguations.

In these rules emerging from the sample pair PREP VERB, the fixed point is called *fixpoint* and the position that is to be disambiguated is called *disambiguation point*. A rule matches the sentence only if all fixpoints are found (they *cannot* be ambiguous) and there is at least one tag from the disambiguation point on the appropriate position.

Sentences are usually largely ambiguous and it is not very probable that all fixpoints are non-ambiguous. This problem can be solved by *merging* negative n-grams, i.e. by creating more complex disambiguation rules out of the negative n-grams. The merge could be done either automatically, or manually. Merging negative n-grams manually means, in fact, to directly write down the disambiguation rules, see Chapter 5.

Automatic merging of negative n-grams doesn't seem to be very meaningful in the context of the Section 4.5. While writing rules directly for an ambiguous input brings some advantages, a FSM can be simply used to implement negative n-grams.

Nevertheless, several attempts have been made to find out an algorithm that (for any input set of negative n-grams) builds a machine that is able to disambiguate directly. None of the ideas leads to an algorithm that would always produce a machine with the outputs identical to the outputs of the generic disambiguation algorithm — there always exist such sentences that the generic disambiguation algorithm decides *more* than the machine suggested by the algorithm.

Later it was realized that the reason is very simple, mainly after formalizing the main goal via transducers. Note that only *sequential finite-state* transducers bring advantages in far less time-consumption against disambiguation via intersection of FSMs (Section 4.5). Processing a sentence with a non-sequential finite-state transducer leads to an exponential search through the transducer's possible computations and hence non-sequential finite-state transducers bring no advantage as compared to disambiguation via FSMs.

To prove that no sequential finite-state transducer can work instead of the generic disambiguation algorithm, we have to recall some definitions (see [28] and [29]):

Definition 4.6 *Let $A = (Q, \Sigma, \Delta, \delta, I, F)$ be a finite-state transducer such that for each word $v \in \Sigma^*$ it outputs at most one word $w \in \Delta^*$. Then the function $f : \Sigma^* \to \Delta^*$ is* realized *by A whenever $f(u)$ is undefined for each word $u \in \Sigma^*$ that is not accepted by A and $f(u) = w$ for each word u that is accepted by A with output w.*

Definition 4.7 *Let Σ, Δ be two alphabets. Let $f : \Sigma^* \to \Delta^*$ be a function that maps the words over the alphabet Σ to the words over the alphabet Δ. The function f is* rational *whenever there exists a finite-state transducer $A = (Q, \Sigma, \Delta, \delta, I, F)$ such that A realizes the function f. The function f is* sequential *such that there exists a sequential transducer $A = (Q, \Sigma, \Delta, \delta, I, F)$ that realizes f.*

Theorem 4.8 *([28]) Let f be a rational function mapping Σ^* to Δ^*. Then f is sequential if and only if there exists a positive integer K such that the following condition holds:*

$$(\forall u \in \Sigma^*)(\forall x \in \Sigma)(\exists w \in \Delta^*)(|w| \leq K \wedge f(ux) = f(u)w). \tag{4.9}$$

Theorem 4.10 *Let B be a finite set of negative n-grams. Then there exists no sequential transducer $A = (Q, \mathcal{P}(T), \mathcal{P}(T), \delta, I, F)$ (using columns-notation) with the following features:*

- *A refuses an input sentence S if and only if B-total-conservative disambiguation of S contains an empty column of tags.*
- *A outputs B-total-conservative disambiguation of an input S whenever S is accepted by A.*

Proof. Let $T = \{t_1, t_2\}$ be the tagset and let $B = \{[t_2, \{t_1, t_2\}, t_1]\}$ be the set consisting of (the only one) negative bigram. Let S_n and S'_n be two sentences:

$$\begin{aligned} S_n &= \{t_1, t_2\}^n \\ S'_n &= \{t_1, t_2\}^{n-1}\{t_1\} \end{aligned}$$

The sentence S_n contains n columns $\{t_1, t_2\}$ and the sentence S'_n contains $n-1$ columns $\{t_1, t_2\}$ and n-th column contains only one tag t_1.

The generic disambiguation algorithm G can do the following: for the sentence S_n, it can delete all readings that contain t_2 followed (not necessarily immediately) by t_1. In particular, the readings of the form $t_1^k t_2^{n-k}$ (for k from 0 to n) will remain in the result. Hence the B-total-conservative disambiguation of S_n is again S_n. In the case of the sentence S'_n, the machine G will delete all readings containing t_2 on any position, hence the only reading will remain — the one consisting of all t_1's.

For a contradiction, let A be the sequential transducer meeting the conditions of Theorem 4.10. The transducer A (for accepted sentences) returns exactly one output, hence there exists a rational function $f : \mathcal{P}(T)^* \to \mathcal{P}(T)^*$ realized by A. Denote $a = \{t_1, t_2\}$ and $b = \{t_1\}$. The function f (due to its definition) maps the sequences a^n to a^n and $a^n b$ to b^{n+1}.

Since A is sequential, f is also sequential. From Theorem 4.8 it follows that there exists an integer $K > 0$ such that the formula (4.9) holds. Take $u = a^K$ and $x = b$.

Now $f(ux) = b^{K+1}$ and $f(u) = a^K$. Hence there is no w such that $f(ux) = b^{K+1} = a^K w = f(u)w$. A contradiction. □

4.5 Effective application of negative n-grams

This section concerns an effective way of applying a set of negative n-grams to an input sentence. Let T be a tagset, B a finite set of negative n-grams and S a sentence to be processed. It has been already shown (cf. [5]) that given a triple (B, T, S) it is NP-hard to determine whether S is B-correct.

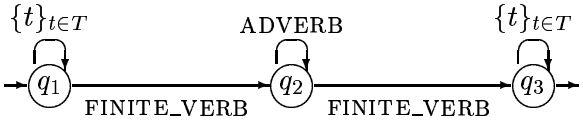


Figure 4.1: FSM generated from the negative bigram b

In practice, the real task is to process a lot of sentences with fixed set B, i.e. it is possible to preprocess the set B in any (even exponential) time to achieve a very quick machine for processing corpora. Section 4.4 also shows that it is not possible to encode negative n-grams into a finite-state transducer that directly outputs a partial disambiguation identical to the one produced by the generic disambiguation algorithm.

However, it is still possible to apply negative n-grams effectively to get B-total-conservative disambiguations. The rest of this section gives formal description of this technique.

In a few words, the set B will be compiled into one (large) finite-state machine $\mathcal{A}_B$. When processing a corpus, each sentence S is again encoded into a FSM and the intersection of S and $\mathcal{A}_B$ is the target B-total-conservative disambiguation (in readings-notation).

A single negative n-gram b from B is a regular expression and it can be translated into a FSM very trivially (see [3]). To accept all $\{b\}$-incorrect sentences, all that remains to be done is to insert cycles over the whole tagset T to all initial and final states of these automata. For example, let the tagset T contain at least two tags FINITE_VERB and ADVERB and let b be our sample negative bigram from Section 4.1:

FINITE_VERB {ADVERB} FINITE_VERB.

The FSM generated from the bigram b is presented in Figure 4.1. The machine contains three states q_1, q_2 and q_3. The state q_1 is the initial state, q_3 is the final (accepting) state. The input alphabet is the tagset T. The machine accepts exactly those readings that *match* the negative bigram b.

Let $\mathcal{A}$ be the union of machines that has emerged from all n-grams in B — then $\mathcal{A}$ accepts all readings that are B-incorrect.

Now let $\bar{\mathcal{A}}$ be a FSM that represents the complement of $\mathcal{A}$ with respect to "everything", i.e. to the machine accepting T^*. Then $\bar{\mathcal{A}}$ accepts only B-correct readings. It is obvious that the machine $\bar{\mathcal{A}}$ can be generated independently of the particular input sentence, hence the complexity of its generation (and even determinization and minimization) is constant and theoretically unimportant in the disambiguation process[4].

[4]In practice, the size of the automaton $\bar{\mathcal{A}}$ (for any more complex set B) is far higher than a size of any "reasonable" sentence automaton and this becomes very significant in the real disambiguation.

It remains to show how to apply the machine $\bar{\mathcal{A}}$ to the input sentence S. The sentence $S = w_1 \dots w_n$ (where $w_i \subseteq T$ for all i from 1 to n) is translated into a machine $\mathcal{S}$ that will accept all readings of the sentence S. The machine contains $n+1$ states $q_0, \dots, q_n$ with the initial state q_0 and the accepting state q_n. The transition from q_i to q_{i+1} (for all i from 0 to $n-1$) is present for all t such that $t \in w_{i+1}$. Note that the machine $\mathcal{S}$ is deterministic.

Since the machine $\bar{\mathcal{A}}$ accepts all B-correct readings and $\mathcal{S}$ accepts exactly all readings of the sentence S, the intersection $\mathcal{I} = \bar{\mathcal{A}} \cap \mathcal{S}$ accepts a reading r if and only if r is both B-correct and a part of S. Hence the machine $\mathcal{I}$ represents the B-total-conservative partial disambiguation of S in readings-notation. It is also possible to reconstruct the B-total-conservative disambiguation of S in columns-notation, see Section 4.6.

Finding the intersection $\mathcal{I}$ of two FSMs is a polynomial task (see [3]). The decision whether there exists an accepting state that can be reached from the initial state in a FSM is also a polynomial task, see the algorithm in Figure 4.2. The algorithm cycles at most $|Q|$ times and each cycle consumes at most $|Q^2 \times T|$ steps. If there exists an accepting state q of $\mathcal{I}$ such that q can be reached from the initial state, then there exists a B-correct reading r of S and S is B-correct.

Hence the detection of B-correctness of an input sentence S with the automaton $\bar{\mathcal{A}}$ fixed is a polynomial task (by the length of S).

4.6 Reconstruction of a disambiguated sentence

This section is a direct continuation of the previous section, including the formalism used there. It describes a reconstruction of B-total-conservative disambiguation in columns-notation from the intersection of the sentence automaton and the n-grams' automaton.

The reconstruction can be made in two steps. In the first step, all states q of the machine $\mathcal{I}$ such that no final state can be accessed from q are removed and a new machine $\mathcal{J}$ is created from the rest. This step is implemented by the algorithm in Figure 4.2. Please note that (since $\mathcal{I}$ accepts exactly the paths of length n) the machine $\mathcal{J}$ contains no cycles and all paths that start in the initial state finish in the accepting state exactly after n transitions. The second step takes the machine $\mathcal{J}$ and creates the B-total-conservative disambiguation of S in columns-notation — it is implemented by the algorithm in Figure 4.3. The algorithm consumes at most $n \cdot |Q|^2 \cdot |T|$ steps. Hence also the B-total-conservative disambiguation in columns-notation can be retrieved in polynomial time.

Formally, if the input sentence S is B-incorrect then the algorithm in Figure 4.2 finishes with an "error-message" and in this case the sentence with all positions empty could be returned as the B-total-conservative disambiguation of S.

A good question (for corpora maintainers) is whether it is practical to convert the machine $\mathcal{I}$ back to the tag sets, since this leads to the loss of information, see Section 3.4.

Input: FSM $\mathcal{I} = (Q, T, \delta, I, F)$ — the intersection of sentence machine $\mathcal{S}$ and the machine $\bar{\mathcal{A}}$ representing the set B of negative n-grams;

Output: non-deterministic machine $\mathcal{J}$ derived from $\mathcal{I}$ by removing all states q such that no state from F can be accessed from q; the corresponding transitions are removed together and treated as undefined; the algorithm also decides whether the sentence S is B-correct;

Variables: V — set of states such that an accepting state is accessible from it; V' — a temporary variable;

1. Set $V = F$, $V' = \emptyset$;
2. If $V \neq V'$ then go to step 3; if $V = V'$ then stop with the following result:
   - if $I \cap V \neq \emptyset$ then S is B-correct, and now let $\mathcal{J} = (V, T, \delta', I \cap V, F)$ where δ' is a restriction of δ only to the states from V, i.e. all transitions (q, x, q') such that either $q \notin V$ or $q' \notin V$ are marked undefined;
   - if $I \cap V = \emptyset$ then the sentence is B-incorrect and any further processing of the sentence must stop here;
3. Let $V' = V$;
4. For all $q \in Q$ and $x \in T$ put q into V whenever there exists a state $p \in V'$ such that $(q, x, p) \in \delta$;
5. Continue at step 2.

Figure 4.2: Removal of garbage states from the machine $\mathcal{I}$

Input: the sentence $S = w_1 \dots w_n$ where $n = \texttt{length}(S)$, the tagset T, the intersection machine $\mathcal{J} = (Q, T, \delta, I, F)$ created from the machine $\mathcal{I}$ by the algorithm in Figure 4.2;

Output: B-total-conservative partial disambiguation $S' = w'_1 \dots w'_n$ of the sentence S;

Variables: V — the set of states accessed so far; V' — the backup of V;

1. Set $i = 1$, $w'_1 = w'_2 = \dots = w'_n = \emptyset$, $V = \{q_0\}$, $V' = \emptyset$;
2. If $i \leq n$ then go to step 3; otherwise stop and return $w'_1 \dots w'_n$;
3. Let $w'_i = \{t; (t \in w_i) \wedge (\exists q \in V, q' \in Q)((q, t, q') \in \delta)\}$;
4. Let $V' = V$ and $V = \{q; (\exists q' \in V', x \in T)((q', x, q) \in \delta)\}$;
5. Let $i = i + 1$;
6. Continue at step 2.

Figure 4.3: Reconstruction of the tag sets of the B-total-conservative disambiguation in columns-notation from the intersection machine $\mathcal{J}$

4.7 Weak points of negative n-grams

Negative n-grams are a very efficient tool in the language modeling, but they have also several disadvantages:

- they look only at one tag at most for each word, i.e. they cannot use the whole morphological analysis (i.e. ambiguity classes) of words; however, looking at the ambiguity classes and using columns-notation brings a lot of problems that are hard to investigate, see Chapter 5;
- any actions behind the definition are impossible (roughly speaking, any non-regular checks, e.g. counting the number of occurrences of some feature in the sentence), see motivation in Section 3.5;
- formula (4.1) that defines negative n-grams is very simple and transparent; however, it is extremely difficult to describe some (even regular) features of the language — for example, grammatical agreement in the gender category will lead to tens of n-grams and the agreement in more categories even leads to hundreds of n-grams; moreover, using tags directly in the definition of n-grams would require detailed understanding of the particular tagset, which reduces the potential number of linguists who are able to write n-grams and obscures the linguistic idea behind the n-gram;
- one of the big advantages of rule-based methods (including negative n-

> grams) is the availability of the debugging process[5]; applying the negative n-grams through the finite automata (i.e. via the only effective way known to get the total and conservative disambiguation) obscures the debugging information.

A relevant question (that should be, however, asked in the theoretical linguistics) is how negative n-grams can be exhaustive in describing a natural language grammar. Of course, in practice it could be assumed that there exists a maximal sentence length in every language and that there is only a finite word set repertory used in the language. Then negative n-grams (where n is the maximal sentence length) are always able to distinguish between correct and incorrect sentences. This is, on the other hand, (still in practice) impossible mainly since working with the language performance data is unrealistic, no matter how large corpora can be maintained with existing computational power. So the question should rather sound as follows:

> How large a piece of grammar can be described by a set of "reasonable" negative n-grams?

Here "reasonable" negative n-grams denote such n-grams that are based on the knowledge of the entire language system (and hence it can be expected that they will be applicable to more than one sentence).

Up to now, it is known that grammatical phenomena that are based on counts of some features in the sentence cannot be described by negative n-grams. In Czech, for example, the number of word forms *se* can be matched with the number of all verbs, verbal adjectives and verbal nouns and the result of such a comparison can be used for disambiguation.

The completion of the list of language phenomena that cannot be described by such negative n-grams is the task for theoretical linguistics.

This is also the question whether some debug information can be retrieved from the automata (i.e. from the intersection of the sentence automaton and the n-grams' automaton). Currently no investigation has been made in this field.

[5]The basic useful information for every n-gram is the information whether (and where) it matches the input sentence.

Chapter 5

LanGR formalism

This chapter formally defines surface grammatical rules and investigates their several important properties — see Section 5.1.

Then the formalism *LanGR* is introduced — Section 5.2 summarizes the main features of the formalism that distinguish it from other programming languages.

Section 5.3 gives a tutorial lesson in writing grammatical rules in the *LanGR* formalism.

The last Section 5.4 concerns relations between the rules written in *LanGR* and the formal definition of surface grammatical rules.

5.1 Surface Grammatical Rules

This section defines surface disambiguation rules (i.e. rules that perform shallow disambiguation) and investigates their features.

The aim of the rules is the maximal reduction of the ambiguity of an input sentence — in the ideal case, to keep only one tag for each word or to mark up the sentence as grammatically incorrect. To achieve this goal, the disambiguation rules must cooperate. In the case of negative n-grams, the cooperation of rules has been implicitly encoded into the finite-state machines, see Chapter 4. While a lot of rules written in *LanGR* so far are in fact negative n-grams, there exists also a number of rules that cannot be translated into negative n-grams — and it is necessary to assume that their power is really equal to a deterministic Turing transducer. However, this assumption entails a lot of problems in the mathematical theory behind the rules — starting with the impossibility to ensure that the output of the disambiguation process will be deterministic.

Let T be a tagset. In this section, any sentence S will be given either in columns-notation, i.e. $S = \{h_i\}_{i=1}^{n}$ where $n = \texttt{length}(S)$ and h_i is a subset of T for all i from 1 to n, or in dispersed-notation, i.e. $S \subseteq T \times \mathbb{N}$. For two sentences S' and S of the same length, let $S' \subseteq S$ denote the fact that S' is a (partial) disambiguation of S[1].

[1]This is natural in dispersed-notation, but it is good to adopt the operator $\subseteq$ also in the columns-notation.

5.1.1 A disambiguation rule

The previous experience with disambiguation rules that perform shallow disambiguation shows that every rule consists of two parts:

1. *configuration* part defines the sequence of positions that must be present in an input sentence to allow the application of the executive part of the rule;

2. *executive* part defines an action that has to be done whenever the configuration part matches the input; usually, the executive part deletes some tags from the sentence.

An executive part of a rule can be very complex and it is generally assumed that its expressive power must be equal to the power of a Turing machine.

On the other hand, configuration parts (at least in the rules written in *LanGR* so far) are regular expressions in fact — the configurations are defined similarly to the negative n-grams, see the expression (4.1), but unlike negative n-grams, the rules must operate on ambiguous inputs.

Let us start with an example — consider the following rule:

> Let n_1, a, n_2 be a sequence of three positions such that the positions n_1 and n_2 are safe nouns (i.e. they contain only nominal tags) and the position a contains at least one adjectival tag.
>
> In this configuration, look whether n_1 and adjectival tags from a can agree in morphological features number, gender and case (i.e. there exists a tag t_1 on position n_1 and an adjectival tag t_2 on position a such that the tags t_1, t_2 share the same values of number, gender and case). Then look whether n_2 and a can agree in number, gender and case.
>
> If neither n_1 nor n_2 can agree with adjectival tags from a then do nothing. If both n_1 and n_2 can agree with adjectival tags from a then do nothing. If only n_1 can agree with adjectival tags from a then remove from a all adjectival tags that do not agree with any tag from n_1. Similarly proceed in case only n_2 can agree with a.

This rule expresses the fact that (in Czech) whenever an adjective stands between two (safe) nouns then the adjective must be an attribute of one of the nouns[2]. As the attributes must agree with the appropriate nouns then if the adjective agrees with exactly one of the nouns then it must be the attribute of the particular noun.

In the rule, the first paragraph represents the *configuration part* that consists of three subsequent positions — safe noun, possible adjective and safe noun. The next two paragraphs define the *executive part* of the rule — it is performed whenever the configuration is found in an input sentence.

Let us formally define the configuration of a rule. Any configuration is a simple regular expression:

[2]The exceptions are extremely rare.

Definition 5.1 (Configuration) *Let T be a tagset and let $X = \mathcal{P}(T)$. Then*

- *any set $c \subseteq (X \setminus \{\emptyset\})$ is a* static (configuration) component*;*
- *if c is a static component then c^* is a* stretching (configuration) component*;*
- *an element c is a* (configuration) component *whenever c is either a static or a stretching component;*
- *if $c_1, c_2, \ldots, c_n$ (for an integer $n > 0$) are components then the sequence $c_1 c_2 \ldots c_n$ is a* configuration.

Definition 5.2 (Empty position) *Let T be a tagset. Let $S = \{h_i\}_{i=1}^n$ be a sentence of length n ($h_i \subseteq T$ for all i from 1 to n). A position k (for k from 1 to n) is an* empty position *whenever $h_k = \emptyset$.*

In other words, a position is *empty* whenever it contains no tags.

Definition 5.3 (Match proposal) *Let $m > 0$ be an integer. A* match proposal of length m for n components *is any non-decreasing sequence $\{j_i\}_{i=1}^m$ such that $0 \leq j_i \leq (n+1)$ for all i from 1 to m.*

An index $i \in \{1, \ldots, m\}$ is outside *the match proposal whenever either $j_i = 0$ or $j_i = n + 1$ hold. Otherwise, the index i is* inside *the match proposal.*

Definition 5.4 (Match) *Let T be a tagset. Let $C = c_1 \ldots c_n$ be a configuration and let a sequence $o = \{j_i\}_{i=1}^m$ be a match proposal of length m (for some $m > 0$) for n components. Let $S = \{h_i\}_{i=1}^k$ be a sentence ($h_i \subseteq T$ for all i from 1 to k). The sequence o is a* match *of the configuration C in the sentence S if the following conditions hold simultaneously:*

- *$k = m$, $0 \leq j_0 \leq 1$ and $n \leq j_m \leq n + 1$;*
- *for every i from 1 to n such that c_i is a static component of C the formula $(\forall \ell \in \{1, \ldots, m\})(j_\ell = i \Rightarrow h_\ell \in c_i)$ holds;*
- *for every i from 1 to n such that c_i is a stretching component of C (and $c_i = c^*$ for some $c \subseteq \mathcal{P}(T)$) the formula $(\forall \ell \in \{1, \ldots, m\})(j_\ell = i \Rightarrow h_\ell \in c)$ holds;*

In this thesis, components of configurations are usually not defined as sets of columns of tags. The components are rather defined as simple formulae that must hold for a particular sentence position to be compatible with the configuration.

For example, a static component c can be defined as *safe verb* — i.e. a position can match the component c whenever it contains verbal tags only. Similarly, a position can match a component *possible singular* if there is at least one tag with the morphological feature "number" set to singular in the position.

Let us demonstrate the terms defined above by an example. Let T be a tagset that consists of five tags: VERB, NOUN_s, NOUN_p, ADVERB, PUNCTUATION.

Let the subscript s denote that a noun is in singular number (p for plural number).

Let $C = c_1c_2c_3$ be a configuration defined briefly by the following table:

c_1	static	safe verb	{VERB}
c_2	stretching	safe adverb	{ADVERB}
c_3	static	safe noun	{NOUN_s, NOUN_p}, {NOUN_p}, {NOUN_s}

The formal definition of tag sets that match every component are present in the last column of the table. Match proposals have still no connection with the example, however, consider three match proposals:

$$o_1 = 1, 2, 3, 4, 4, 4, 4$$
$$o_2 = 0, 0, 0, 0, 1, 3, 4$$
$$o_3 = 0, 0, 1, 2, 2, 3, 3$$

Consider the following sentence (morphologically unambiguous in the tagset T):

	Rozvazuje	snaživě	provázek	,	nevidí	konec	.
	verb	adverb	noun_s	punct	verb	noun_s	punct
o_1	1	2	3	4	4	4	4
o_2	0	0	0	0	1	3	4
o_3	0	0	1	2	2	3	3

In the sentence, two matches of the configuration C can be found — the first one on the positions 1–3 (o_1) and the second one on the positions 5–6 (o_2). The match proposal o_3 is not the match of C in the sentence, because the third position *provázek* with index 1 in o_3 is not a safe verb (required by c_1).

In the match o_1, the positions *Rozvazuje snaživě provázek* are inside the match and the rest of sentence is outside the match. In the match o_2, the positions *nevidí konec* are inside the match and the rest of sentence is outside the match. In other words, the positions inside a match are those that meet the configuration.

It is a well-known fact that if C is a configuration (i.e. a simple regular expression) then there can be constructed a deterministic finite-state machine that verifies whether an input sentence S contains a match of C. For the following formal sections, however, the situation is much more simple — it is expected that a sentence to be processed is equipped with a match proposal and the machine has to decide whether the proposal is a match of the configuration in the sentence.

Definition 5.5 (Match verificator) *Let T be a tagset and $X = \mathcal{P}(T)$. Let $C = c_1 \ldots c_n$ be a configuration. Let $J = \{0, 1, \ldots, n+1\}$ and let U be a finite set of integers such that $J \subseteq U$. The* match verificator *for the configuration C is a deterministic finite-state machine $A = (Q, X \times U, \delta, \{q_0\}, F)$ where $Q = \{q_0, \ldots, q_n\}$. The transition (partial) function $\delta : Q \times (X \times U) \to Q$ and the set F of accepting states is defined by the following algorithm:*

1. *let δ be undefined in its entire domain; let $G = \{q_0\}$ be a temporary set of states; for all $h \in X$ define $\delta(q_0, (h, 0)) = q_0$;*

2. *for every $i = 1, \ldots, n$ do the steps 3–4; then for all $q \in G$ and for all $h \in X$ define $\delta(q, (h, n+1)) = q$, define $F = G$ and stop;*

3. *if $c_i \subseteq (X \setminus \{\emptyset\})$ is a static component, then for all $q \in G$ and all $h \in c_i$ define $\delta(q, (h, i)) = q_i$; let $G = \{q_i\}$; continue with next i in step 2;*

4. *if c_i is a stretching component such that $c_i = c^*$ for some $c \subseteq (X \setminus \{\emptyset\})$ then for all $q \in G$ and for all $h \in c$ define $\delta(q, (h, i)) = q_i$ and $\delta(q_i, (h, i)) = q_i$; insert q_i into G; continue with next i in step 2.*

Let us note that the definition of the function δ is not ambiguous in the steps 3 and 4, because in step i of the algorithm the state q_i is never an element of G and the input elements change in every step (with increasing i).

The set U in Definition 5.5 is present only to simplify the formal definition of the disambiguation process, see Section 5.1.2. The set J could be used directly instead of U for the time being.

The order of states in the set $Q = \{q_0, \ldots, q_n\}$ is significant in Definition 5.5. In fact, an input match proposal specifies the sequence of states the match verificator enters during its operation.

Definition 5.6 (Executive transducer) *Let T be a tagset and $X = \mathcal{P}(T)$. Let $C = c_1 \ldots c_n$ be a configuration. Let $J = \{0, 1, 2, \ldots, n+1\}$. Let U be a finite set of integers such that $J \subseteq U$. Let Σ be a working alphabet such that there exist working tape boundaries $\tau, \chi \in \Sigma$. Let $M = \{\leftarrow, \cdot, \rightarrow\}$. A deterministic linear Turing transducer $E = (Q, X \times U, X, \Sigma, \delta, \{q_0\}, F)$ is an* executive transducer *for the configuration C whenever the following conditions hold simultaneously:*

- *E accepts any input;*
- *the transducer E uses only the matching subsequence for its operation, i.e. its operation on the pair (h, ℓ) is independent of particular $h \in X$ for every ℓ such that $\ell < 1$ or $\ell > n$; formally: for any integer i such that $i < 1$ or $i > n$, for any $h \in X$ and for all pairs $(q, \sigma) \in Q \times \Sigma$ there exists at most one quintuple $(q_1, m_1, \sigma_1, m_2, \alpha_1) \in Q \times M \times \Sigma \times M \times X^*$ such that for all 8-tuples $(q, (h, i), \sigma, q', m', \sigma', m'', \alpha') \in \delta$ the following formula holds*
$$(q', m', \sigma', m'', \alpha') = (q_1, m_1, \sigma_1, m_2, \alpha_1).$$
- *for any input $\{(h_j, i_j)\}_{j=1}^m$ the transducer E outputs a sequence $\{k_j\}_{j=1}^m$ such that $k_j \subseteq h_j$ for all j from 1 to m.*

The last point of the definition ensures that any executive transducer performs partial disambiguation of an input sentence.

Definition 5.7 (Disambiguation rule) *Let T be a tagset and let $X = \mathcal{P}(T)$. Let $C = c_1 \ldots c_n$ be a configuration and let $J = \{0, 1, \ldots, n+1\}$. Let U be a finite set of integers such that $J \subseteq U$. Let A be a match verificator and let E be an executive transducer (both for the configuration C).*

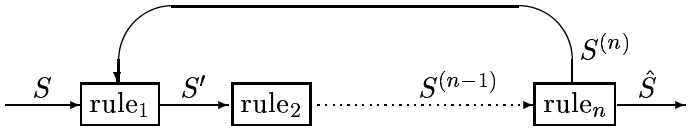


Figure 5.1: Cooperation of disambiguation rules on an input S

A deterministic Turing transducer D is a disambiguation rule *for the configuration C and for the executive transducer E whenever it operates in the following steps on any input sequence S of elements of $X \times U$:*

1. *D simulates the operation of A on S;*
2. *if A accepts S then D simulates the transducer E on S and outputs the output of E;*
3. *if A doesn't accept S then D outputs S;*
4. *D accepts S.*

5.1.2 Disambiguation process

No single disambiguation rule can decide every ambiguity that can appear in input sentences. The basic idea of the rule-based disambiguation is that every rule deals with a single more or less complex phenomenon of the grammar and all the rules *cooperate* to remove as many tags as possible.

A natural way of making the rules cooperate is to run the rules as a "serial combination" in a loop until nothing changes in the sentence. In other words, previous rules can remove tags such that their absence makes the application of the later rules possible. This simple model is illustrated in Figure 5.1 — every rule takes the output of the previous one as its input. The first rule starts with an original sentence S on its input. During the process, the sequence of partial disambiguations of S is created: $S', S'', \ldots, S^{(n-1)}, S^{(n)}, \ldots$ such that $S^{(k+1)}$ is always a partial disambiguation of $S^{(k)}$ (it is possible that $S^{(k)} = S^{(k+1)}$). Whenever any of n rules deletes a tag, the output of the last rule is passed to the first one again. The loop terminates with the partial disambiguation $\hat{S}$ if none of the rules deletes a tag.

A formal definition of the disambiguation process, however, is more complex.

Lemma 5.8 *Let $m > 0$ and $n > 0$ be two integers. There exists at most $(n+2)^m$ different match proposals of length m for n components.*

Proof. A match proposal is a variation of $n+2$ integers of length m. □

In fact, the total number of different match proposals is lower, because every proposal is non-decreasing. In this thesis, however, it is only important that the set of all match proposals is finite.

Let D be a finite set of disambiguation rules. In the rest of the thesis it is assumed that all disambiguation rules use the same tagset (usually denoted by T). It is also assumed that there exists an integer Ξ such that a configuration of any disambiguation rule contains at most Ξ components and that every disambiguation rule expects match proposals for Ξ components on its input[3]. These assumptions are natural, but they must be stated explicitly to simplify the following theorems.

Definition 5.9 (Disambiguation process) *Let T be a tagset and let $X = \mathcal{P}(T)$. Let D be a finite set of rules. Let S be an input sentence of length n.*

Let P be the set of all match proposals of length n for Ξ components. Let $\prec$ be an ordering on the set $D \times P$, i.e. $D \times P = \{(d_i, p_i)\}_{i=1}^{|D\times P|}$ such that $(d_i, p_i) \prec (d_{i+1}, p_{i+1})$ for every i from 0 to $|D \times P| - 1$. Then $\prec$ is a disambiguation ordering *of the sentence S by the rules D.*

The disambiguation process *of the sentence S by the rules D with ordering $\prec$ is defined by the following algorithm:*

1. *let $S^0 = S$ be the first iteration of disambiguation of S; let $j = 0$;*
2. *let $i = 1$; let $S_0^j = S^j$;*
3. *if $i > |D \times P|$ then let $j = j + 1$ and let $S^j = S^{j-1}_{|D\times P|}$; if $S^j = S^{j-1}$ then stop and output S^j; otherwise, continue by step 2;*
4. *if $i \leq |D \times P|$ then take the pair $(d_i, p_i) \in D \times P$; let the rule d_i operate on the input $\{(h_\ell, j_\ell)\}_{\ell=1}^n$ where $S_{i-1}^j = \{h_\ell\}_{\ell=1}^n$ and $p_i = \{j_\ell\}_{\ell=1}^n$; store the output to S_i^j; let $i = i + 1$ and continue by step 3.*

The disambiguation process ensures that every pair from $D \times P$ is applied at least once after the last change has been made to the sentence and before the algorithm stops.

Lemma 5.10 *Every disambiguation process is finite.*

Proof. There is only a finite number of tags in every sentence and the maximum number of iterations of j is equal to the number of tags in S. The set $D \times P$ is finite due to Lemma 5.8. □

The choice of the disambiguation ordering, however, can be significant for the result of the disambiguation process — see Section 5.1.4.

Definition 5.11 (Trivial rule) *A disambiguation rule d is* trivial *whenever for any input sentence S the disambiguation process of S by $\{d\}$ outputs S. Otherwise, the rule is* non-trivial.

[3]Of course, rules based on configurations that contain less than Ξ elements never accept a match proposal outside the range defined by the configuration.

5.1.3 Grammar checking

This section investigates the properties of sentences that contain *empty positions*, see Definition 5.2 on page 41.

Empty positions play a very important role in processing corpora with disambiguation rules. Usually, a sentence that enters the disambiguation process contains no empty columns (because every position is assigned all adequate tags from the morphological analyzer[4]). If a position of the sentence loses all of its tags during the disambiguation process then something goes wrong either in the sentence or in the rules (see Goal 3.1 on page 22 and the subsequent paragraphs). On the other hand, an empty position does not directly indicate that the particular position is "incorrect" — the problem can be found elsewhere in the sentence. Every empty position should be subject to further investigation of the sentence and of the rules. The list of rules that has been applied to the sentence is the most valuable information for the investigation.

Because an empty position can appear in an arbitrary step of the disambiguation process, it is necessary to pay attention to the operation of the rules on empty positions, i.e. to the definition of transitions over empty sets of tags. Otherwise, the rules can completely destroy the sentence and the critical information about the first empty position (and the corresponding list of rules) is lost. This is the reason for excluding empty positions from configurations of rules, see Definition 5.1 on page 41. Let us highlight the assets of this approach:

- whenever an empty position appears in the sentence during the disambiguation process, no rule can use it inside a match; roughly speaking, the adjacent positions of this empty position are not "destroyed" and the sentence can be investigated to find out what is "wrong";
- it does not forbid the application of the rule outside the empty position; if there is no empty position inside the match then the rule can be applied.

Definition 5.12 (Correct sentence) *Let D be a finite set of disambiguation rules. Let S be a sentence. The sentence S is* D-correct *whenever for every disambiguation ordering $\prec$ of S by D there is no empty position in the output of the disambiguation process of S by D with $\prec$. Otherwise, the sentence S is* D-incorrect.

In other words, a sentence S is D-incorrect in two cases: either it contains an empty position, or there exists its disambiguation S' by D such that S' contains an empty position.

Proposition 5.13 *Let D be a finite set of rules. For every D-correct sentence S there exists a partial disambiguation $S' \subseteq S$ such that S' is D-incorrect.*

Proof. Let S' contain an empty position. □

[4]Note that it depends on how unknown words are handled by the analyzer — see Section 7.4 for the particular situation in Czech.

Definition 5.14 (Grammar checker) *Let D be a finite set of disambiguation rules such that there exists a D-incorrect sentence without empty position. Then the set D is a* grammar checker.

The only objective of Definition 5.14 is that the set D is able to mark up a sentence as incorrect. A disambiguation rule d is a *grammar checker* whenever the set $\{d\}$ is a grammar checker.

Proposition 5.15 *There exists a non-trivial disambiguation rule such that it is not a grammar checker.*

Proof. Let $T = \{t_1, t_2\}$ be a tagset. Let $C = \{\{t_1, t_2\}\}$ be a configuration of a single static component $\{\{t_1, t_2\}\}$. Let A be a match verificator for C. Let E be an executive transducer for the configuration C such that it always disambiguates $\{t_1, t_2\}$ to $\{t_1\}$ (i.e. E deletes the tag t_2 from any ambiguous input position). It is obvious that the disambiguation rule d for C and E never produces an empty position, i.e. it is not a grammar checker.

Let $S = \{h_i\}_{i=1}^{n}$ be a sentence without an empty position ($h_i \subseteq T$ for all i from 1 to n). Let S contain at least one ambiguous position j and let $P = \{p_i\}_{i=1}^{n}$ be a match proposal such that $p_j = 1$, $p_i = 0$ for all $i < j$ and $p_i = 2$ for all $i > j$. In this situation, the disambiguation rule d for C and E deletes t_2 from the position j. The rule d is non-trivial. □

From the proof of Proposition 5.15 it can be seen that the proposition is not tagset-dependent. Thus for any tagset (with at least two tags) a disambiguation rule can be constructed such that it is able to delete a tag, however, it is not a grammar checker.

All negative n-grams are grammar checkers, but they require a new definition to operate on ambiguous input and also an additional definition of the executive part. The complete technique is not presented here.

For illustration, for a negative bigram PREP FINITE_VERB the following configuration can be constructed: let T be a tagset and let T_{P} and T_{FV} denote the sets of all prepositional and finite-verbal tags, respectively. The configuration $C = c_1 c_2$ contains two static components: c_1 contains all tag sets h such that $h \subseteq T_{\mathrm{P}}$; c_2 contains all tag sets h such that $h \cap T_{\mathrm{FV}} \neq \emptyset$. Now C matches all pairs of positions such that the first one is a *safe preposition* and the second one is a *possible finite verb*.

The executive part of the constructed rule can be defined trivially to remove all finite-verbal tags from the second position.

Such a rule deletes all tags from a safe finite verb preceded by a safe preposition, i.e. it is a grammar checker.

5.1.4 Order-independent rules

It has already been said that the order of disambiguation rules during the disambiguation process can be significant for the output of the process. This section investigates the properties of disambiguation rules that can avoid this order-dependency.

Definition 5.16 (Strongly order-independent rules) *A set D of disambiguation rules is* strongly order-independent *whenever for every input sentence S there exists a partial disambiguation $S' \subseteq S$ such that for all disambiguation orderings $\prec$ (of S by D) the sentence S' is the output of the disambiguation process of S by the rules D with the ordering $\prec$.*

Definition 5.17 (Order-independent rules) *A set D of disambiguation rules is* (weakly) order-independent *whenever the following conditions hold simultaneously:*

- *for any D-incorrect sentence S and for any disambiguation ordering $\prec$ (of S by D) the output of the disambiguation process of S by D with $\prec$ contains an empty position;*
- *for every D-correct input sentence S there exists its partial disambiguation S' such that for all disambiguation orderings $\prec$ (of S by D) the sentence S' is the output of the disambiguation process of S by D with $\prec$.*

For both strongly and weakly order-independent sets of rules it is ensured that the results of all disambiguation processes (for all orderings of rules) are identical for any D-correct input.

The difference between Definitions 5.16 and 5.17 is that (weakly) *order-independent* rules can be ambiguous on "ungrammatical" inputs while *strongly* order-independent rules always produce "deterministic" output. Both strongly and weakly order-independent rules, however, deterministically distinguish between D-correct and D-incorrect sentences. Strongly order-independent sets of rules are also weakly order-independent.

In the rest of the thesis, strongly order-independent sets of rules are not used — these sets could be very useful in the applications outside the framework of this thesis, e.g. in automatic grammar correcting (the correction of an error can be proposed in dependence of what has happened in the sentence during the operation of rules).

Definition 5.18 (Safely based rule) *Let T be a tagset. A disambiguation rule d is* safely based *whenever the following condition holds for every sentence $S = \{h_i\}_{i=1}^{n}$ ($h_i \in \mathcal{P}(T)$ for all i from 1 to n), for every match proposal $p = \{j_i\}_{i=1}^{n}$ and for every partial disambiguation $S' = \{h'_i\}_{i=1}^{n}$ of S ($h'_i \subseteq h_i$ for all i from 1 to n) such that $h'_i \neq \emptyset$ holds for every i such that $j_i \geq 1 \wedge j_i \leq n$:*

> *Let $O = \{o_i\}_{i=1}^{n}$ be the output of the operation of d on the input $\{(h_i, j_i)\}_{i=1}^{n}$ and let $r_i = h_i \backslash o_i$ for all i from 1 to n. Let $O' = \{o'_i\}_{i=1}^{n}$ be the output of the operation of the rule d on the input $\{(h'_i, j_i)\}_{i=1}^{n}$. Then the formula $(\forall i \in \{1, \ldots, n\})(o'_i \cap r_i = \emptyset)$ holds.*

Put simply, let S be a sentence and let R be a set of tags deleted from S. A safely based rule must delete at least R from any partial disambiguation S' of S (if the particular match proposal doesn't contain empty position inside).

The general idea behind this definition is that any safely based rule should operate (on a correct input) in the same way whenever it gets more information (i.e. when more tags are deleted in the input).

Theorem 5.19 *Every finite set of safely based rules is order-independent.*

Proof. Let D be a finite set of safely based disambiguation rules.

First, let S be a D-correct sentence of length n. Let us assume, for a contradiction, that there exist two different disambiguation orderings $\prec_1$, $\prec_2$ (of S by D) such that the outputs of the disambiguation processes of S by D with orderings $\prec_1$ and $\prec_2$ are not identical. Let P be the set of all match proposals of length n.

Let $D^i = (d^i_1, p^i_1)(d^i_2, p^i_2)\dots(d^i_{k_i}, p^i_{k_i})$ (for $i = 1, 2$ and some integers $k_1, k_2 > 0$; where $(d^i_j, p^i_j) \in D \times P$ for all $i = 1, 2$ and $j = 1, 2, \dots, k_i$) be the complete sequence of rules and match proposals that have been applied during the disambiguation processes of S by D with the ordering $\prec_i$; i.e. every pair from $D \times P$ can appear more times both in D^1 and in D^2. Let $S^i_0, S^i_1, \dots, S^i_{k_i}$ (for $i = 1, 2$) denote the sequence of partial disambiguations of S during the disambiguation processes, i.e. $S^i_0 = S$ and the sentence S^i_ℓ is the output of the operation of d^i_ℓ on the input $\{(h_j, p_j)\}^n_{j=1}$ where $S^i_{\ell-1} = \{h_j\}^n_{j=1}$ and $p^i_{\ell-1} = \{p_j\}^n_{j=1}$ (for every ℓ from 1 to k_i).

From now on, let us use the dispersed-notation of sentences. S is D-correct, hence neither $S^1_{k_1}$, nor $S^2_{k_2}$ contains an empty position. Let $(t, j) \in S$ denote a tag t on a position j such that $(t, j) \in S^1_{k_1} \setminus S^2_{k_2}$ (t is present in $S^1_{k_1}$ and not in $S^2_{k_2}$ on position j; the opposite situation is analogous). Hence there exists an index i such that $(t, j) \in S^2_i$ and $(t, j) \notin S^2_{i+1}$ (i.e. (t, j) has been deleted by the pair (d^2_{i+1}, p^2_{i+1})).

If $S^1_{k_1} \subseteq S^2_i$, then the pair (d^2_{i+1}, p^2_{i+1}) had to delete (t, j) also from $S^1_{k_1}$ (because it is safely based and due to the definition of the disambiguation process) — a contradiction and $S^1_{k_1} \cap (S \setminus S^2_i) \neq \emptyset$ must hold. Hence there exists $(t', j') \in S^1_{k_1} \setminus S^2_i$. Again, let $i' < i$ be an index such that $(t', j') \in S^2_{i'}$ and $(t', j') \notin S^2_{i'+1}$. Since the formula $(t', j') \in S^1_{k_1}$ holds, the inequality $S^1_{k_1} \cap (S \setminus S^2_{i'}) \neq \emptyset$ must hold. By induction, the sequence of integers $i, i', i'', \dots$ is decreasing and the formula $S = S^1_{k_1} \cap (S \setminus S^2_0) \neq \emptyset$ holds. A contradiction with $S^2_0 = S$.

Now, let S be a D-incorrect sentence. If S contains an empty position, the output of any disambiguation of S also contains an empty position. Let S contain no empty position. Hence there exists a disambiguation ordering $\prec_1$ such that the disambiguation process of S by D with $\prec_1$ produces an empty position.

It remains to be shown that for any other ordering $\prec_2$ the disambiguation process of S by D with $\prec_2$ produces an empty position. For a contradiction, let us assume that the output of the disambiguation with an ordering $\prec_2$ contains no empty position.

Let $(d^i_1, p^i_1), \dots, (d^i_{k_i}, p^i_{k_i})$ be the complete list of rules and match proposals that have operated in the disambiguation process of S by D with $\prec_i$ (for $i = 1, 2$ and for some integers k_1, k_2). Let $S^i_0, \dots, S^i_{k_i}$ be the sequence of partial disambiguations of S during the disambiguation process of S by D with $\prec_i$ (for $i = 1, 2$).

Due to the assumptions, there exists an index i such that S^1_i doesn't contain an empty position and S^1_{i+1} contains an empty position. If $S^2_{k_2} \subseteq S^1_i$ then the

pair (d^1_{i+1}, p^1_{i+1}) should also empty the position in $S^2_{k_2}$, hence there exists a pair $(t,j) \in S^2_{k_2} \setminus S^1_i$. Let $i' < i$ be an index such that (t,j) is present in $S^1_{i'}$, but (t,j) is not present in $S^1_{i'+1}$. If $S^2_{k_2} \subseteq S^1_{i'}$ then the rule $d^1_{i'+1}$ should also delete (t,j) from $S^2_{k_2}$. Hence there exists $(t',j') \in S^2_{k_2} \setminus S^1_{i'}$. By induction, the sequence of indexes $i, i', i'', \ldots$ is decreasing and the formula $S^2_{k_2} \cap (S \setminus S^1_0) \neq \emptyset$ holds. A contradiction. □

Proposition 5.20 *There exists a disambiguation rule that is not safely based.*

Proof. Let us recall the rule d from the proof of Proposition 5.15 on page 47, the input sentence $S = \{(t_1,1),(t_2,1)\}$ and its partial disambiguation $S' = \{(t_2,1)\}$. The sentence S contains only one (ambiguous) position with two tags t_1, t_2. It is obvious that the rule d deletes the tag t_2. On the other hand, d deletes nothing from S'. □

Let $D = \{d\}$ be a set consisting of the single rule d from the proof of Proposition 5.15 and let S be an arbitrary sentence (using the tagset $T = \{t_1, t_2\}$). It is obvious that the disambiguation process of S by D (with any disambiguation ordering) changes all ambiguous positions $\{t_1, t_2\}$ in S to $\{t_1\}$ (and it does nothing more). Hence D is an order-independent set of rules.

Corollary 5.21 *There exists an order-independent set of rules that contains a rule that is not safely based.*

Theorem 5.22 *Let d be a safely based disambiguation rule such that there exists at least one input sentence $S = \{h_i\}_{i=1}^n$ and a match proposal $p = \{p_i\}_{i=1}^n$ such that d deletes a tag from $\{(h_i, p_i)\}_{i=1}^n$. Then d is a grammar checker.*

Proof. Let us assume that d deletes a pair (t,j) from S (in dispersed-notation) during its operation on the input $\{(h_i,p_i)\}_{i=1}^n$. Let S' be a partial disambiguation of S such that it contains only a single tag t on position j (and all other positions are equal to the original ones in S; i.e. $S' = \{h'_i\}_{i=1}^n$ where $h'_i = h_i$ for all $i \neq j$ and $h'_j = \{t\}$). There is no empty position inside the match p in S. Hence there is no empty position inside the match proposal p in S'. Because d is safely based, it must delete (t,j) from S' and produce an empty position. Hence d is a grammar checker. □

The Theorem 5.22 offers a simple way of marking up the rules that are not safely based — if a rule d can disambiguate and it is not a grammar checker, then d is not safely based.

Proposition 5.23 *There exists a grammar checker such that it is not safely based.*

Proof. Let us modify the rule d from the proof of Proposition 5.15 so that d deletes all tags from the ambiguous position (i.e. it rewrites $\{t_1, t_2\}$ to $\emptyset$). The rule d is obviously a grammar checker, but it cannot delete anything from any non-ambiguous sentence. □

Definition 5.24 (The preservation of incorrectness) *A set D of disambiguation rules* preserves incorrectness *if every partial disambiguation S' of a sentence S is D-incorrect whenever S is D-incorrect.*

A rule d *preserves incorrectness* whenever the set $\{d\}$ preserves incorrectness.

Theorem 5.25 *Any finite set of safely based rules preserves incorrectness.*

Proof. Let T be a tagset and let $D = \{d_1, \ldots, d_n\}$ be a set of safely based rules. Let S be a D-incorrect sentence of length n. For a contradiction, let $S' \subseteq S$ be a D-correct sentence (hence neither S nor S' contains an empty position).

From the fact that S is D-incorrect it follows that there exists a list of rules and match proposals $(d_1, p_1), \ldots, (d_k, p_k)$ that have been applied during the disambiguation process of S by D (with any ordering, because D is order-independent, see Theorem 5.19) such that no empty position has been present until (d_k, p_k) is applied. Let $S_0, S_1, \ldots, S_k$ be the sequence of partial disambiguations of S such that $S_0 = S$ and S_i is the output of operation of d_i on $\{(h_j, p_j)\}_{j=1}^n$ where $S_{i-1} = \{h_j\}_{j=1}^n$ and $p_i = \{p_j\}_{j=1}^n$. Let r be the index of the empty position in S_k (if there are more empty positions, let r be the first one).

If $S' \subseteq S_{k-1}$ then d_k would produce empty position r while operating on S' and the proposal p_k. Hence there exists a pair $(t, j) \in S' \setminus S_{k-1}$. Let (t, j) be deleted during the disambiguation process of S by a rule $d_{k'}$ and a proposal $p_{k'}$ for some $k' < k$. If $S' \subseteq S_{k'-1}$ then (t, j) would not be present in S' (because of the application of the rule $d_{k'}$ on S' with the proposal $p_{k'}$). Hence $S' \cap (S \setminus S_{k'-1}) \neq \emptyset$. By induction, there is a decreasing sequence of integers $k, k', k'', \ldots, k_\ell$ such that $k_\ell = 1$ and $S' \cap (S \setminus S_0) \neq \emptyset$. A contradiction — S' contains an empty position. □

There exists a rule such that it preserves incorrectness, but it is not safely based — just take any non-safely-based rule without the grammar-checking property (see Proposition 5.15 on page 47).

Proposition 5.26 *There exists an incorrectness-preserving grammar checker such that it is not safely based.*

Proof. Let $T = \{t_1, t_2, t_3, t_4\}$. Let d be a disambiguation rule with the following properties:

- if an unambiguous position with the tag t_3 is immediately preceded by a position that contains only tags from the set $\{t_1, t_2\}$ then delete t_3;
- if an unambiguous position with the tag t_4 is immediately preceded by an ambiguous position $\{t_1, t_2\}$ then delete t_1 from the ambiguous position.

In other words, the rule d applies exactly to the following pairs of positions (the particular tag deleted by d is marked up in a box):

$$\begin{array}{l} t_1 \\ t_2 \end{array} \boxed{t_3} \;\Big|\; t_1 \; \boxed{t_3} \;\Big|\; t_2 \; \boxed{t_3} \;\Big|\; \boxed{t_1} \; t_4 \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad t_2$$

The rule d can produce an empty position (by deleting t_3 in the appropriate context), i.e. d is a grammar checker. The rule d also preserves incorrectness, because the only incorrect sentences are those with an unambiguous position with t_3 preceded by either $\{t_1, t_2\}$, $\{t_1\}$ or $\{t_2\}$. On the other hand, d is not safely based, because d deletes nothing from the sequence $\{t_1\}\{t_4\}$ while it deletes $\{t_1\}$ from the sequence $\{t_1, t_2\}\{t_4\}$. □

5.2 Key features of *LanGR*

This section illustrates basic features that must be present in any programming language to be comfortably used by linguists oriented to shallow syntax.

5.2.1 The architecture of *LanGR*

A completely new programming language *LanGR* has been developed as a language for defining functions (or predicates, if you want) that allow linguists to write down disambiguation rules with all possible comfort. Programming in *LanGR* is separated into two levels — *core* and *application*.

The core level of *LanGR* is a programming language similar to `C` or `Pascal`. It separates linguists from implementation problems and introduces several innovative features that reflect specific properties of linguistic programming (see below). The expressive power of programs written in the core has been identified with that of a deterministic Turing transducer, which makes it possible to implement anything "effectively enumerable"[5]. The core level of *LanGR* has been completely described in [2].

It is unknown whether the expressive power of deterministic Turing transducer is really necessary to write down the set of disambiguation rules that covers a sufficiently large part of Czech grammar. On the other hand, it seems that finite-state transducers are not powerful enough (see Sections 4.4 and 4.7).

Let's highlight basic features of the core level of *LanGR*:

- the independence of character sets of the input/work files (*LanGR* works internally in UNICODE, see [33]);
- in most cases, no brackets are necessary to specify arguments in expressions — predicates can consist of more than one keyword and the arguments are separated by blanks, which increases the readability of programs and rules by humans;
- *LanGR* automatically searches for unknown identifiers, i.e. mostly nothing like annoying `#include` command is required;
- *LanGR* allows for a better tracing of rules' execution (in comparison with negative n-grams realized by a FSM, for example);
- *LanGR* makes it possible to define useful shortcuts (e.g. `Singular` is defined to refer to all tags with singular interpretation independently of a particular part-of-speech value).

The really necessary features of *LanGR* core — first-order formulae and "parallel" computations — are presented in the subsections with further description.

A majority of functions directly used by linguists is not a part of the core. This set of functions and definitions is usually understood as the application level of *LanGR*. The application level of *LanGR* also contains all natural language dependent definitions, hence rewriting the core is not necessary to use *LanGR* for another natural language.

[5] This idea formulated as a hypothesis is expressed by Church-Turing thesis, see [31].

The application functions are described partially in Section 5.3, more exhaustive list is available in Appendix B. However, the list of application functions is continuously updated.

5.2.2 Columns-notation in *LanGR*

It has been mentioned that the rules in *LanGR* use columns-notation of sentences. The good question is why the rules use this "weaker" description of sentence. The answer is immediate, but it is based on ideas rather than on facts and proofs.

First of all, let as recall how readings-notation is used with negative n-grams — a sentence is translated into a FSM (i.e. it is stored in the readings-notation), then it is intersected with the automaton representing the rules and the result is the disambiguated sentence (again in the readings-notation).

Once the rules become non-regular (e.g. context-free or beyond), the intersection of the rules with the sentence automaton gets outside FSM — and hence the nice representation of sentence in readings-notation is lost.

LanGR is designed to implement rules that are far beyond regular expressions. So this simple application of rules is inaccessible.

Moreover, a lot of rules attempt to look at the ambiguity classes of words in the sentence, which is not natural in readings-notation (and pretty natural in columns-notation).

As a consequence, the decision to implement columns-notation in *LanGR* has been made without further investigation.

5.2.3 First-order formulae

The *LanGR* formalism (and its compiler) introduces real availability of *first-order formulae* — including logical conjunctions, predicates and quantifiers. This section tries to explain why such a very powerful instrument is necessary for writing the rules.

Historically, boolean expressions seemed to be essential to express relations between particular tags in different positions, see [11]. These relations should "simulate" advantages of readings-notation in *LanGR*. This feature of *LanGR* has not been used so far.

On the other hand, first-order formula can be used to naturally describe the conditions stated by linguists in rules. Let's illustrate it by the following rule:

> If a noun stands as the last word in a sentence following immediately a safe preposition with the locative case valency only, then the noun must be in the locative case.

The rule in fact says that there is a *configuration* of three subsequent items:

- safe preposition with the locative case valency only,
- possible noun,
- end of sentence.

Here *preposition with the locative case valency only* and *noun* refer to the morphological tags. However, since the words in the particular sentence can still be ambiguous, it is necessary to specify whether these sub-formulae have to hold for *all* or *at least one* tag in the positions — this is expressed by the predicates *safe* and *possible*.

With a little abstraction, the first two references to positions can be rewritten as

- $(\forall t \in T_1)(\texttt{preposition}(t) \wedge \texttt{locative}(t))$
- $(\exists t \in T_2)(\texttt{noun}(t))$

where T_1 and T_2 refer to the set of all remaining tags of the penultimate and final position in the sentence, respectively. Predicates $\texttt{preposition}(t)$, $\texttt{local}(t)$ and $\texttt{noun}(t)$ are used to determine some features of a tag t — they can be translated into *LanGR* with minimal effort.

When a configuration matches the sentence, *executive* part of the rule should be performed — in this case, all non-locative noun tags can be deleted from the set T_2. These tags that must be deleted can be also specified by a boolean expression:

$$\texttt{noun}(t) \wedge \neg\texttt{locative}(t).$$

To illustrate how *LanGR* handles the above mentioned expressions, let's present a part of the code corresponding to the above rule:

```
ITEM IsSafe Preposition and Locative;
subs = ITEM Possible Noun;
ITEM SentenceEnd;

LEAVE ONLY Locative or not Noun IN subs;
```

The command `ITEM` verifies whether a position in the input sentence matches the formula it carries — in the first line of the code, the formula is `IsSafe Preposition and Locative`. `ITEM` simply applies this formula to the current position in the data and if the result of the test is positive, the rest of the configuration is processed.

The keyword `IsSafe` specifies that *all tags* on the particular position (supplied by `ITEM`) must match the rest of the line, in this case, `Preposition and Locative`. In other words, `IsSafe` stands for the quantifier $\forall t \in Y$, where the set Y is specified by `ITEM`.

In fact, *LanGR* collects the formulae in the reverse order, mainly since the domain of the quantifiers is known only at run-time. Hence sub-formulae on single tags are translated into a boolean expression (`Preposition and Locative`), which is passed to the higher function (`IsSafe`) that adds the quantifier and the result is then passed to `ITEM`.

This approach allows for a large variability in defining new functions — functions that define features of single tags, or functions that add quantifiers, and many more (unification, selection of subsequence of positions...).

Also, it is very simple to add new "trivial" predicates that are used in single-tag formulae. For example, there could appear a lemma-match "predicate" like `lemma == "být"` or `lemma member of AdjectivesWithGenitiveValency`.

The independence on the particular tagset also partially follows from the use of the first-order formulae — the predicates (e.g. `Preposition`) for matching a single tag can be simply redefined for another tagset (see Section 5.3.12). On the other hand, implementation of the first-order formulae considerably slow down the whole computation.

5.2.4 Branches of program execution

Section 5.1.1 introduces a disambiguation rule — a non-deterministic Turing transducer. Section 5.1.2 defines the disambiguation process, i.e. an algorithm that performs the disambiguation of an input sentence by a set of rules. This section describes the way of how *LanGR* implements the disambiguation process.

Let $\mathcal{D}$ be a finite set of disambiguation rules. For the disambiguation process the order of rules in the set $\mathcal{D}$ is irrelevant. The order of match proposals used by the rules is also irrelevant. The disambiguation process only ensures that all the rules and all the proposals are executed from the last modification of the sentence before the algorithm stops.

This is exactly what the *LanGR* interpreter does. A user (a linguist) selects a single rule or a group of rules (see Sections 5.3.11 and 6.1.1) to be applied to the input. During the processing, the order of rules is fixed by the definition of the group being selected. The order of rules can be, however, significant for the result of disambiguation, see Section 5.1.2.

The non-trivial task is to ensure that all possible match proposals are used to process the input. In practice, it is impossible to verify all possible match proposals for an input sentence. Hence every match verificator (i.e. a finite-state machine) must be replaced by a non-deterministic finite-state *transducer* that returns the matches of the configuration in the sentence. In this case, the implementation must be able to process all the non-deterministic branches of the transducer's operation.

LanGR implements the `split` command that allows for splitting the program execution into "parallel" branches — every transition from the `split` command is assigned separate program execution branch and every branch is stored on a stack of branches. The interpreter has to process all the branches on the stack, however, the order of the branches on the stack can be significant for the result of disambiguation.

Shortly, the `split` command contains a list of command sequences — every sequence initiates a separate branch (i.e. one of the non-deterministic transitions). The command-flow of every branch continues after the `split` command (see [2]). The *LanGR* interpreter processes the branches in a well-defined order — from the first command sequence in the `split` command to the last one[6].

[6] As `split` commands can be nested, the interpreter performs *pre-order backtracking* of the tree of all branches, in fact.

Hence *LanGR* interpreter is able to implement the disambiguation process with both the fixed order of rules and the fixed order of matches.

5.3 Tutorial rules

This section introduces several basic types of surface grammatical rules that can be written in the *LanGR* formalism. It is not possible to generate the full list of functions that are available for linguists to express their ideas (at least since every linguist can define his own special functions he needs in the rules).

However, this tutorial is an essential tool for a *LanGR* beginner. It starts with simple n-gram rules and later proceeds to more complex rules with unification, obscure configurations and so on. It also gives hints on grouping rules and writing "nice" comments and examples into the source files.

The tutorial does not investigate technical details — where to place source files of the rules and how to execute the rules. This information can be found in Chapter 6 and in [2].

Trivial note: The *LanGR* formalism itself doesn't define in any way how "nicely" rules should be written. Nor can it verify linguistic correctness of the rules[7]. *LanGR* is just a way of writing disambiguation rules with all available comfort.

5.3.1 Negative bigram

The most trivial rule formalizes a simple negative bigram of two subsequent finite verbs:

```
/*(1)*/  rule TwoFiniteVerbs { // start of the rule

/*(2)*/     rulevariant VarI {
/*(3)*/        ITEM IsSafe FiniteVerb;
/*(4)*/        ToDelete = ITEM Possible FiniteVerb;
/*(5)*/        DELETE FiniteVerb FROM ToDelete;
/*(6)*/     };

/*(7)*/     rulevariant VarII {
/*(8)*/        ToDelete = ITEM Possible FiniteVerb;
/*(9)*/        ITEM IsSafe FiniteVerb;
/*(10)*/       DELETE FiniteVerb FROM ToDelete;
/*(11)*/    };

/*(12)*/ };
```

First of all, every program code (including the rules which are also part of a program written in *LanGR*) should be sufficiently commented. *LanGR* supports comments in the rules in the way similar to C++ or Java programming

[7] *LanGR* interprets the *core* level of itself; all grammar knowledge is stored in the *application* level.

languages — comments can be either enclosed in the special character sequences `/* */` (as shown above in the line numbers) or put after `//` to the end of line (as shown in line `1`).

Let's go through the rule line by line. The first line initiates the rule by the keyword `rule`. Its name is `TwoFiniteVerbs` (*LanGR* is case-insensitive, hence it is equal to `twofiniteverbs`) and its content is enclosed in curly brackets on lines `2`–`11`.

Every rule must contain one or more *variants* — each variant is introduced with the keyword `rulevariant` followed by its name (`VarI` or `VarII` in this example) and the content of the variant.

The first variant `VarI` implements the negative bigram of two finite verbs where the first one is a safe finite verb (i.e. all tags of the word represent finite verbs) and the second one is a possible finite verb (i.e. it can be ambiguous between finite verb and non-finite verb).

The variant `VarII` implements the reverse order of the elements, i.e. a possible finite verb is followed by a safe finite verb.

In line `3`, the keyword `ITEM` specifies that a single position (word) of the sentence is required to match this point of the configuration. The argument of `ITEM` (the rest of the line, in this case) is the condition that the position must meet to participate in the configuration: `IsSafe FiniteVerb`. Here `IsSafe` specifies that all tags on this position must match the rest of the condition — the rest of the condition is `FiniteVerb` which determines a property of a single tag.

Put simply, a word of an input sentence matches line `3` whenever all of its tags represent a finite verb.

Line `4` specifies the disambiguation point and it must be somehow remembered (since there can be more disambiguation points in a rule variant, generally) for later processing in the executive part of the rule.

This is implemented by assigning the whole `ITEM` to the variable `ToDelete`. Formally, the variable `ToDelete` is a local integer variable — if the `ITEM` on line `4` matches a position of the input sentence, the variable `ToDelete` will contain the index of this position (indexed from 1). Any name can be used instead of `ToDelete`, e.g. `x`, `i` and so on. However, the identifier must not collide with any global identifier or with a locally valid identifier, see [2].

Now, lines `3` and `4` form the *configuration* part of the variant `VarI`. Since there is nothing more to search in the sentence, the variant can proceed with the *executive* part on line `5`.

The keyword `DELETE` enables the linguist to remove some tags from a position — in this case, all tags that match the condition `FiniteVerb` are deleted from the position `ToDelete`. The keyword `FROM` is part of the syntax of `DELETE` and separates the condition from the index specification.

The variant `VarII` is only a mirror of `VarI` so it seems no further explanation is necessary.

Please note that two subsequent `ITEM`s match only two adjacent positions in the sequence (of course, if their respective conditions hold for the particular positions). No position can be between these positions — this will be illustrated later in another example.

Note also that the core of *LanGR* very strictly requires the correct syntax of the rules — e.g. every command must be ended with a semicolon (e.g. `rule`, `rulevariant`, `ITEM`, `DELETE`) and curly brackets are also mandatory as shown in the example.

LanGR offers no special command to separate the configuration and executive parts of the rule. This is because these parts are obviously separated by the particular commands they employ.

Every rule by default (via the function `RuleStart`, see Appendix B) starts with the command `Any SEQUENCE` that ensures that the interpreter will look for the configuration in the whole sentence (i.e. it doesn't stop after the first match, but it tries to find all the sub-sequences of the sentence that match the configuration). The command `Any SEQUENCE` is exactly the same as `SEQUENCE` of any items, see Section 5.3.2.

5.3.2 Negative bigram with context

In Czech, the word form `mnohem` is very often immediately followed by an adjective or adverb — in this configuration, the second position must be in the comparative degree.

Moreover, the form `mnohem` can be separated from the adjective/adverb by a sequence of other words — adverbs (different from the word `mnohem`) in the positive degree.

```
rule MnohemComp {
   RuleVariant mcvar {

/*(1)*/   ITEM IsSafe (lemma == "mnohem");

/*(2)*/   SEQUENCE OF IsSafe
          (Adverb and (lemma != "mnohem") and
          (not (Comparative or Superlative)));

/*(3)*/   adjadv = ITEM IsSafe (Adjective or Adverb);

/*(4)*/   REPORT "If the word form "
          emphasize("mnohem")
          " is followed by a sequence of adverbs in the positive"
          " degree then the subsequent adjective or adverb "
          emphasize(WordFormOnPosition(adjadv))
          " must be in the comparative degree.";

/*(5)*/   LEAVE ONLY (Comparative) IN adjadv;

   }; // end of variant
}; // end of rule
```

The configuration part of the rule consists of lines 1, 2 and 3. Line 1 matches any word with the safe lemma `mnohem`. The condition `lemma == "mnohem"` is

again the test of one tag and `lemma` is one of the categories of the tag (see Section 5.3.12). Since `lemma` is a string field of the tag, the operator `==` here compares two strings, `lemma` (the lemma of the particular tag just being processed) and `"mnohem"`.

Line `2` matches any sequence of positions such that all their tags meet the condition

```
Adverb and (lemma != "mnohem")
   and (not(Comparative or Superlative)).
```

Logical conjunctions `and`, `or`, `not` and parentheses need no further explanation. The operator `!=` (inequality) is just a negation of the equality test `==`. Hence the conditions hold for all adverbial tags that do not contain the lemma `mnohem` and that are in the positive degree (i.e. not in the comparative or superlative one).

The sequence on line `2` includes also empty sequences, i.e. if lines `1` and `3` match adjacent positions then the configuration is matched.

Line `3` simply matches all positions with only adverbial or adjectival tags (including combination of both).

Line `5` is the executive part of the rule. The command `LEAVE ONLY` is the negation of `DELETE` — in this case, only comparative tags will remain on the position `adjadv`.

Line `4` represents one of the features that are available in *LanGR* and inaccessible in the negative n-grams — the debugging information. The command `REPORT` generates a string from its arguments which is remembered and used by the executive part whenever a tag is deleted. The string contains further description of what happened. `REPORT` can take any number of arguments of any data type.

Let's briefly explain the argument for the particular command `REPORT` on line `4`. The function `emphasize` takes a string argument and returns again a string — if it is possible (i.e. if the output device allows it), the result is highlighted (usually printed in the italics). The function `WordFormOnPosition(x)` returns a word form on position `x` of the sentence.

The rest of arguments are string constants that are copied to the generated string without any change.

5.3.3 Basic unification

This section introduces basic unification command `UNIFY`. Unification, roughly speaking, means the matching of the corresponding morphological features (categories) in two positions of the sentence.

Assume `pos1` and `pos2` are two positions matched by the `ITEM` command in the configuration part of a rule. They can be unified, for example, in the `gender` category using the following executive command:

```
UNIFY pos1 WITH pos2 IN [gender];
```

This command compares the `gender` category in the two positions and deletes all tags (from both positions) that do not match any tag from the other position. In this comparison, tags with non-applicable `gender` category match anything.

For example, let `pos1` contain two nominal tags — in the `neuter` and `feminine` gender. The position `pos2` contains two verbal tags — in `masculine` and `neuter`. In this case, unification in gender would keep only `neuter` tags and the other tags will be deleted.

If `pos1` is a safe `preposition` (where `gender` is not an applicable category), it matches everything and nothing happens.

Finally, if `pos1` is a combination of prepositional and nominal (e.g. `neuter` and `feminine`) tags and `pos2` contains again two verbal tags (`masculine` and `neuter`) then the prepositional tag from `pos1` will match anything from `pos2` and hence `pos2` will be left untouched. However, nominal tags from `pos1` must match the tags from `pos2` and hence `feminine` will be deleted from `pos1`. Preposition in `pos1` will remain, of course.

The list of categories in the `UNIFY` command must be always enclosed in square brackets. Whenever there is more than one category in the list, the categories must be separated by commas.

The following rule illustrates the practical use of the unification command:

```
rule prepnperspron {
   RuleVariant v1 {

      prepos = ITEM Possible Preposition;

      npron = ITEM IsSafe PronounNPersonal;

      REPORT "A preposition must stand in front the"
         " personal pronoun "
         emphasize(WordFormOnPosition(npron))
         ".";

      LEAVE ONLY Preposition IN prepos;

      REPORT "The preposition "
         emphasize(WordFormOnPosition(prepos))
         " must match the case of "
         emphasize(WordFormOnPosition(npron))
         ".";

      UNIFY prepos WITH npron IN [case];

   }; // of rule variant
}; // of rule
```

In Czech, a personal pronoun[8] starting with the letter *n* must always be preceded by a preposition. The rule checks whether such a safe pronoun exists in the sentence (position `npron`) and in this case it removes all non-prepositional tags from `prepos` (by the `LEAVE` command).

[8] In the third person, i.e. except for *nás*, *nám*, *námi*.

Then the unification of `prepos` and `npron` is performed. It is clear that the unification must be applied *after* removing non-prepositional tags from `prepos` — these tags could, as it happens, unify with the tags from `npron` causing tags which otherwise would be deleted to remain in `npron`.

Thus, the executive part of this rule contains two commands — `LEAVE` and `UNIFY`. Please note that `REPORT` command composes a (globally defined) report string, which is used as the debug string in the executive part till the end of the rule variant or till another `REPORT` command. For example, in the rule above the first `REPORT` applies to the tags that are deleted by the `LEAVE` command, while the second `REPORT` applies to the tags that are deleted by the `UNIFY` command.

Special attention should be paid to the identifier `PronounNPersonal`. The list of n-starting pronouns, e.g. `(form == "něho" or form == "němu" or ...)`, could be used instead of this identifier. Since it can be expected that this list will be used also in other variants and rules it is useful to store it into a separate definition and refer to it by a single identifier (`PronounNPersonal`, in this particular case) — writing such definitions requires, however, basic skills in using the core level of *LanGR* and it will be described in Section 5.3.4.

5.3.4 Maintenance of word lists

In the rule-based methods, particular word forms or lemmas are often used, since a lot of words have special features in every natural language and not all these features are available in tags. It can be also often seen that some word forms or lemmas are used together (in the same combination) in several different rules.

Hence whenever a rule uses a set of words that is larger than two or three words or that is frequently used also in other variants or rules then it is a good idea to enclose this set into a special definition — a global variable. The following example illustrates this by a simple rule:

```
charset "unix";

/* lemmas of the word forms "aby", "kdyby", ... */
shared Vocabulary AbyKdybyLemmata;

AbyKdybyLemmata = [
"aby",
"kdyby"
];

rule AbyJsi {
   RuleVariant v1 {

      condtvar = ITEM IsSafe (Conditional or lemma member of
                  AbyKdybyLemmata);
      jsemjsi = ITEM Possible ((lemma == "být")
                  and (Present or Future
```

```
                    or Conditional or Imperative));

      REPORT "Conditional word form "
        emphasize(WordFormOnPosition(condtvar))
        " cannot be followed immediately by the form "
        emphasize(WordFormOnPosition(jsemjsi))
        " of the verb " emphasize("být") "!");

      DELETE ((lemma == "být")
                  and (Present or Future
                     or Conditional or Imperative))
             FROM jsemjsi;

   }; // end of variant v1
}; // end of rule
```

The line `shared Vocabulary AbyKdybyLemmata;` introduces the `shared`[9] variable `AbyKdybyLemmata` of the data type `Vocabulary` (i.e. set of strings) into the program.

The following line defines the set itself as a list of two strings `aby` and `kdyby`. The list is (as well as any other set definition) enclosed in square brackets and separated by commas. The definition is a simple assignment of the set to the variable `AbyKdybyLemmata` by the assignment command `=`.

Please note that the expression

```
lemma member of <something>
```

can always be replaced with

```
lemma == "<something1>" or lemma == "<something2>" or ...
```

where `<something1>`, `<something2>` and the rest are exactly the strings stored in the list `<something>`. Large expressions, however, are confusing and hard to manage and, moreover, sets are much effectively processed in comparison with expressions.

If the set of word forms or lemmas is used in more than one rule, it is useful to put them into a separate file. In this case, the file looks as follows:

```
/* AbyKdybyLemmata contains lemmas for words "aby",
 * "kdyby", ...
 */
charset "unix";

shared Vocabulary AbyKdybyLemmata;

AbyKdybyLemmata = [
"aby",
"kdyby"
];
```

[9] The opposite option is `private`, see *LanGR* manual [2] for explanation.

The file name must correspond with its content — by the convention of *LanGR* the file should be named `abykdybylemmata.rlm` (unless several default options of the compiler are violated which is not recommended). In this case the lines `shared Vocabulary AbyKdybyLemmata;` and the definition of the set `AbyKdybyLemmata` should be deleted from the definition file of the rule `AbyJsi` and the compiler itself will find the definition of `AbyKdybyLemmata` automatically in the file `abykdybylemmata.rlm`.

In the rule `AbyJsi`, the command `charset` is also used. Whenever a rule (or any other file in *LanGR*) uses some non-ASCII characters (which is usually the case in any language different from English), the file should start with the command `charset` specifying the character set used in the file.

This is, unfortunately, the simplest solution to the problem with multiple character sets for every language — most common options for `charset` are `"unix"` and `"windows"` which probably requires no explanation. *LanGR* internally works in UNICODE, hence the problem arises only with input and output files. See also the *LanGR* manual [2].

5.3.5 Sequence unification

The rule `PrepGroup` below demonstrates the unification of more than two elements. The name `PrepGroup` stands for "prepositional group", a sequence starting with a safe preposition, ended with a safe noun and some additional words in between. If the configuration is found, all the participating words should be unified in number, gender and case:

```
rule PrepGroup {
   Rulevariant v1 {

      safeprep = ITEM IsSafe Preposition;

      seq = SEQUENCE OF IsSafe ((AdjectiveSynt or PronounSeSi
               or Adverb or Particle
               or Interjection) and (not(AdverbPrep)));

      firstnoun = ITEM IsSafe Noun;

      SEQUENCE OF IsSafe (Adverb or PronounSeSi or Particle
               or Interjection);

      ITEM IsSafe ((Nominative or Vocative)
               or Noun or Pronoun or Numeral
               or Verb or Preposition or Punctuation);

      INSERT safeprep INTO seq;
      INSERT firstnoun INTO seq;

      UNIFY seq IN [number,gender,case];
```

Input: Sequence of positions $w_1, \ldots, w_n$ (in columns-notation). Categories $c_1, \ldots, c_m$.

Output: Sequence of positions $w'_1, \ldots, w'_n$. A position w'_i contains all tags that remain after the unification of positions $w_1, \ldots, w_n$ in categories $c_1, \ldots, c_m$ in position w_i (for i from 1 to n).

Variables: A set C of "under-specified" tags.

1. Set C to the set of all combinations of categories $c_1, \ldots, c_m$ present in w_1.
2. For all $i = 2, \ldots, n$ do step 3. Then continue by step 4.
3. Compare the set C to the position w_i: Let $C' = C$ and let R be a set of "under-specified" tags (for the categories $c_1, \ldots, c_m$) on the position w_i. Let $C = \emptyset$. For every $c' \in C'$ and $r \in R$, use the algorithm in Figure 5.3 to generate an "under-specified" tag $u_{c',r}$ that matches both c' and r. If the tag $u_{c',r}$ exists then let $C = C \cup \{u_{c',r}\}$.
4. For all $i = 1, \ldots, n$ do the following step.
5. Set $w'_i = \emptyset$. For all tags $t \in w_i$: put t into w'_i whenever it unifies with at least one tag from C.

Figure 5.2: Unification of three and more words

```
   }; // end of variant
}; // end of rule
```

The rule `PrepGroup` doesn't use any non-ASCII character — no `charset` command is required.

The configuration part consists of five points — the first three points describe "extended" disambiguation point, the remaining two define an additional condition for the context of the configuration.

The executive part of the rule contains three commands. The command `UNIFY` unifies all the words contained in `seq` in the categories `number`, `gender` and `case`.

The previous `INSERT` commands insert two positions `safeprep`, `firstnoun` into the sequence `seq`. Since `safeprep` and `firstnoun` are simple integers (indexes of positions) and `seq` is a set of integers, there is no technical magic in these insertions.

On the other hand, linguistically the two positions `safeprep` and `firstnoun` are also made members of the unification and hence the whole prepositional group (including these two positions) is unified in number, gender and case. In particular, the preposition `safeprep` is also unified in case with `firstnoun`.

Input: Two "under-specified" tags t, r. Categories $c_1, \ldots, c_m$.

Output: On success, the output is an "under-specified" tag $u_{t,r}$ that matches both t and r. On failure, the output is undefined.

1. Let $u_{t,r}$ be undefined in all categories.
2. For all $i = 1, \ldots, m$ do the following steps.
3. Let v_t and v_r be the value of the category c_i in the tag t and r, respectively. If both v_t and v_r are applicable and $v_t \neq v_r$ then stop — the algorithm failed and the tag $u_{t,r}$ doesn't exist. Otherwise, let $v = v_t$. If v is not applicable, let $v = v_r$;
4. Set the value of the category c_i in $u_{t,r}$ to v.

Figure 5.3: Unification of two "under-specified" tags

The algorithm that unifies more than two positions is semi-formally presented in Figure 5.2. The term "under-specified tags" (for morphological features (categories) $c_1, \ldots, c_m$) in the algorithm denotes tags with the values defined only for categories $c_1, \ldots, c_m$ — the values of the remaining categories are undefined. Two tags unify whenever their values of the categories $c_1, \ldots, c_m$ match. Two values v_1, v_2 of a category c_i match whenever either $v_1 = v_2$ or at least one v_1, v_2 is non-applicable.

A set C in the algorithm can sometimes remain empty (the positions cannot be unified). In this case, all the positions $w_1, \ldots, w_n$ will lose all their tags. The linguists must be aware of this — cf. *conditional* unification later.

The algorithm presented in Figure 5.2 uses a sub-routine (see Figure 5.3) that generates an "under-specified" tag $u_{t,r}$ that matches two input "under-specified" tags t, r. Roughly speaking, the sub-routine generates an "intersection" of the input tags t, r in the categories $c_1, \ldots, c_m$ — in the "intersection", the `n/a` value represents a join (i.e. an upper bound) of any particular value of any category. If there exists a category such that the "intersection" of the tags t, r in this category is empty, the sub-routine fails (the tag $u_{t,r}$ cannot be found).

5.3.6 Selecting a subsequence of positions

This section is the first one that investigates the features of *LanGR* that are outside the expressive power of negative n-grams.

The command `SELECT` allows the user to select a sub-sequence from a sequence (usually specified by the `SEQUENCE` command). It is used whenever a sequence is a part of the disambiguation point, but the disambiguation is to be made only in some particular words of the sequence.

The following rule accounts for the case ambiguity — if there are two safe locatives in the sentence and there is no noun between them then all words in-

between are disambiguated as locatives. However, there are some exceptions...

```
charset "unix";

rule LocMeziLoc {
   Rulevariant v1 {

      ITEM IsSafe ((Noun or Adjective
                    or Pronoun or Numeral or
                    Preposition) and Locative);

      seq = SEQUENCE OF MustNotBe Noun;

      ITEM IsSafe ((Noun or Adjective
                    or Pronoun or Numeral) and Locative);

      selected = SELECT (IsSafe (Adjective or Pronoun
                    or Numeral) and (Nominative or Genitive
                    or Dative or Accusative or Vocative
                    or Locative or Instrumental))
                  and (MustNotBe (PronounSyntNoun
                    or PronounSeSi))
                  and not (lower form Member of NonLocForms)
                FROM seq;

      DELETE (Nominative or Genitive or Dative
                    or Accusative or Vocative or Instrumental)
             FROM selected;

   }; // end of variant
}; // end of rule
```

The `SELECT` command with its condition almost copies the definition `MustNotBe Noun` except for `PronounSyntNoun` and `PronounSeSi` and the list of lemmas in the vocabulary `NonLocForms` (the list of lemmas that can appear inside a locative case sequence with a non-locative case).

Note: The value of `form` is taken over from the input data as is. While `lemma` is (already as the result of morphological analysis) in lowercase characters (unless it is a proper noun and so on), `form` can be capitalized or completely in the uppercase characters (or whatever else). As the lists of forms (`NonLocForms`, in this case) usually contain only lowercase word forms, it is necessary to convert `form` into lowercase before the comparison performed by the `lower` command, e.g. `lower form`.

The `SELECT` function uses the same condition type as `ITEM` or `SEQUENCE` and returns a subsequence (optionally empty) from the input sequence (in this case `seq`). The subsequence can be assigned by the assignment command `=` to a variable (integer set, in this case `selected`).

Since `DELETE` command uses only the `selected` sequence, the exceptions are excluded from the deletion.

It is to be noted that the modifier `MustNotBe` is used in the same manner as `IsSafe` and `Possible`. A position in the sentence matches the condition with the `MustNotBe` modifier whenever no tag in the position meets the rest of the condition.

Rewriting the rule without `SELECT` would be demanding. The number of exceptional word forms in the sentence is unknown and `DELETE ... FROM <sequence>` command itself offers no possibility to exclude some positions of `<sequence>` from the deletion. The only option is to process the set `seq` sequentially index by index and delete each index separately, but this requires special knowledge of the *LanGR* core and it is exactly what the function `SELECT` does.

5.3.7 Conditional unification

The following rule extracts (in its configuration part) a nominal group from the sentence. It starts with a pronoun (with syntactically adjectival function) and ends with a safe noun. If all members of the group can be unified in number, gender and case, then it is safe to perform unification (see the algorithm in Figure 5.2). If the unification cannot be done, nothing is deleted.

```
rule AgreementSyntAdjSubst {
   RuleVariant v1 {

      safesyntpron = ITEM IsSafe
           (lemma Member of PronounSyntAdj);

      seq = SEQUENCE OF IsSafe
           (AdjectiveSynt
              or (lemma Member of PronounSyntAdj)
              or Adverb or Particle);

      firstnoun = ITEM IsSafe Noun;

      INSERT safesyntpron INTO seq;
      INSERT firstnoun INTO seq;

      UNIFY CONDITIONALLY seq
            IN [number,gender,case];

   }; // end of variant v1
}; // end of rule
```

The conditional unification is specified by the keywords `UNIFY CONDITIONALLY` — it takes the same arguments as the simple `UNIFY` command. It initially checks whether all the positions can be unified. If so, the unification is performed (i.e. tags that do not match are deleted as if `CONDITIONALLY` were not present).

If the unification is not possible (i.e. simple `UNIFY` would delete all tags in all positions from `seq`) then the sentence is left untouched.

5.3.8 Unification with restrictions

```
Rule Jehoz {
   RuleVariant v3 {

      PronounJehoz = ITEM IsSafe
         (lower form == "jehož" or PronounRelativePoss);

      SEQUENCE of IsSafe ((Adjective or PronounSyntAdj
            or NumeralSyntAdj)
         and lemma != "všechno"
         and lemma != "ten"
         and lemma != "všichni"
         and lemma != "to");

      Noun1 = ITEM IsSafe Noun;

      ITEM IsSafe not (AdjectiveVerbal);

      REPORT "Vztažné zájmeno přivlastňovací "
        emphasize(wordformonposition(PronounJehoz))
        " se musí shodovat s následujícím substantivem "
        emphasize(wordformonposition(Noun1))
        " v rodě, čísle a pádě!");

      UNIFY UNILATERALLY PronounRelativePoss
        FROM PronounJehoz WITH Noun1
        IN [number,gender,case];

   }; // konec varianty v3
};
```

In Czech, relative possessive pronouns (`jehož`, `jejíž`,...) must unify with the following noun. However, the word form `jehož` is ambiguous between relative possessive pronoun and relative personal pronoun. Here it is appropriate to use one of the unification commands with restrictions. The `UNIFY` command above unifies the positions `PronounJehoz` and `Noun1`, but it has two restrictions:

- it takes into consideration only `PronounRelativePoss` from the position `PronounJehoz` — the tags that do not match `PronounRelativePoss` condition are completely ignored in the unification (and they are never deleted by this rule);
- the keyword `UNILATERALLY` express the wish of the linguist to affect only the first of the two unified positions.

In this case, all tags from `Noun1` are taken into consideration while matching the positions, but no tag from `Noun1` is really deleted. On the other hand, only `PronounRelativePoss` tags are taken into consideration while matching the positions and all such tags that do not match any tag from `Noun1` are deleted.

There are several other variants of unification, see Appendix B.

5.3.9 Sentence boundaries

A lot of rules is based on a fact that their configuration is either at the beginning or at the end of the whole sentence. So it is necessary for the linguists to be able to specify these boundaries in configurations of the rules.

In some systems (cf. [1]), this situation is solved by adding "virtual" positions *SentenceStart* and *SentenceEnd* directly to the data. *LanGR* doesn't follow these approaches — the variable `Text` that represents the sentence in *LanGR* contains exactly the real positions in the text.

Modified `ITEM` commands (`ITEM SentenceStart` and `ITEM SentenceEnd`) can be used to put the configuration at the start/end of the sentence. The commands are very often combined with the sequence of punctuations, e.g.

```
nn = ITEM IsSafe Noun; // 'nn' is the last one in the sentence
SEQUENCE of IsSafe Punctuation;
ITEM SentenceEnd;
```

The command `ITEM` modified by `SentenceStart` or `SentenceEnd` returns nothing, hence the item cannot be stored in a variable.

The sentence boundaries can be also checked by other modifications of the `ITEM` command, see Section 5.3.10.

5.3.10 Reducing the number of variants

This section investigates the variants of the `ITEM` and `SEQUENCE` commands that allow the linguists to reduce the number of rule variants.

```
Rule NounKtery {
   RuleVariant v1 {

      PRE-SENTENCE ITEM IsSafe
        Preposition or Adjective or PronounSyntAdj
        or NumeralSyntAdj or Adverb
        or (Verb and not lemma member of VerbContent)
        or ConjunctionSubordinate;

      SEQUENCE of IsSafe (Adjective
        or NumeralSyntAdj or PronounSyntAdj or Adverb);

      Noun1 = ITEM IsSafe ((Noun or PronounPersonal)
        and (not WordContent));

      ITEM IsSafe Comma;
```

```
      FACULTATIVE ITEM IsSafe Preposition;

      Relat  = ITEM Possible
        (lemma == "který" or lemma == "jaký");

      UNIFY UNILATERALLY Relat WITH Noun1 IN [gender,number];

   };
};
```

The rule `NounKtery` uses the command `FACULTATIVE ITEM`. Unlike plain `ITEM`, the command `FACULTATIVE ITEM` need not be present in the sentence to meet the configuration. If the `FACULTATIVE ITEM` feature were not available in *LanGR*, the variant given above would have to be rewritten into two variants.

Other "shortcuts" to reduce the number of variants are `PRE-SENTENCE ITEM` and `POST-SENTENCE ITEM`. These items match two types of positions — either the sentence boundary[10] or the real position that matches the argument of `ITEM`. In the rule `NounKtery`, the first `ITEM` matches either the start of the sentence (i.e. it operates in the same way as `ITEM SentenceStart`) or the real position that matches the formula

```
IsSafe Preposition or Adjective or PronounSyntAdj
        or NumeralSyntAdj or Adverb
        or (Verb and not lemma member of VerbContent)
        or ConjunctionSubordinate
```

In all commands `FACULTATIVE`, `PRE-SENTENCE` and `POST-SENTENCE ITEM`, it is also possible to store the index of the matched position into an integer variable. If `FACULTATIVE ITEM` is not present in the sentence or if `PRE-SENTENCE ITEM` or `POST-SENTENCE ITEM` match the sentence boundary, the variable is set to `undef` and hence it cannot be used later in the rule (e.g. in the executive commands).

The last simple modified command is `NonEmpty SEQUENCE`. It operates in the same way as simple `SEQUENCE` — it only ensures that the sequence of positions will always match at least one position, e.g.

```
NonEmpty SEQUENCE of IsSafe (Adjective
        or NumeralSyntAdj or PronounSyntAdj or Adverb);
```

Decreasing the number of variants makes it possible to increase the speed of the execution of the rules and, in most cases, enhances understandability of the rules for potential readers.

5.3.11 The grouping of rules

The rules can (and also should) be organized into *groups*. Generally, the groups have been designed to create a tree structure (rules that solve closely related

[10] `PRE-SENTENCE` for the sentence beginning and `POST-SENTENCE` for the sentence end.

problems form small groups that are associated into larger ones and the hierarchy is conventionally ended up with the `root` group that covers all rules).

The syntax of the groups is described completely in the *LanGR* manual, but it is very simple. A new group can be defined as follows:

```
group Agreement {
   SubjectPredicateAgreement; // a subgroup
   AgreementSyntAdjSubst; // single rule
};
```

Every new group is introduced by the keyword `group` followed by the name of the group (`Agreement`, in this example). The content of the group is enclosed in curly brackets. The group contains the semicolon-separated list of elements — each element is a name of either a rule, or a group. Definition of groups must not contain a loop.

Each group should be defined in a separate file with the name corresponding to the name of the group, i.e. `agreement.rlm`, in this example.

Groups are useful for organizing the rules into logical components. However, they are important also in applying the rules. Each group has a tree structure with sub-groups in the non-terminal nodes and with rules in the leaves. The order of leaves is significant — the rules are applied in the order they are written.

The group `root` should contain all rules as its leaves. Hence any new group or rule should be inserted into another already existing group or directly into `root`. Sometimes, it can be useful to create a group outside `root`, e.g. to change the default order of the rules.

5.3.12 Definition of new identifiers

This section presents a tutorial lesson in defining new identifiers that can be used in the rules. This facility requires non-trivial knowledge of the core of *LanGR*, but some types of definitions are used very often in the rules and they are not so complicated.

In the previous sections, two types of new definitions have mostly been used — either a *predicate expression* variable or a *vocabulary* variable. The second type of identifiers has been described in Section 5.3.4. This section investigates predicate expressions.

Let's start with an example: the definition of the `Nominative` identifier. This identifier is used in the condition after `IsSafe`, `Possible` or `MustNotBe` quantifiers as part of `ITEM` or `SEQUENCE`, e.g.

```
ITEM MustNotBe Nominative;
```

From the construction of conditions it follows that `Nominative` must be a condition that is applied to a single tag — in this particular case, this condition should specify that the given tag represents the nominative case.

In the application level of *LanGR*, the tags are represented by the structure data type `InterpretationType`. This structure contains one field for each linguistic category, e.g. fields `number`, `gender`, `PoS` (part-of-speech) and `case`. The definition of `Nominative` uses the field `case`.

The field `case` takes (in the definition of `InterpretationType`) its data type `CaseType` — the list of possible values of `case` can be retrieved from the definition of `CaseType`:

```
type CaseType domain {  // case
Nom,// nominative
Gen,// genitive
Dat,// dative
Acc,// accusative
Voc,// vocative
Loc,// locative
Ins // instrumental
};
```

Immediately, the following comparison is used to check whether a tag contains the nominative case:

```
case == Nom
```

Now the body of the `Nominative` identifier is clear and it can be defined as any other variable similarly to the example in Section 5.3.4:

```
/* nominative */

parse PositionType;

shared predex Nominative;

Nominative = ( case == Nom );
```

The line `parse PositionType;` tells the compiler to immediately process the definition file of the identifier `PositionType`[11]. This is necessary in order to introduce the identifier `case`, which doesn't have a special definition file and hence it cannot be found automatically by the compiler.

The line `shared predex Nominative;` declares a variable `Nominative` with the data type `predex` (first-order formula). The variable `Nominative` is declared as `shared`, i.e. it is shared in all the branches of the program execution. Whenever it is expected that the variable will not change its value during the program execution, it should be declared as `shared`. The opposite keyword to `shared` is the keyword `private`. A `private` declaration is often used to explicitly declare local variables and sometimes also to declare global variables — in all cases, the use of the `private` keyword requires advanced skills in programming in the core of *LanGR*.

In the third line, the variable `Nominative` is assigned a formula that compares a value of the `case` category (of the particular tag that must be later specified) with the constant value `Nom`. `Nom` is a value from the domain of the `case` category.

Similarly all category-checking identifiers can be defined, for example:

[11] `PositionType` is the data structure that stores one position of the sentence.

```
parse PositionType;

shared predex PersonFirst;

PersonFirst = ( person == PFirst );
```

Here `person` is one of the categories (morphological features) and `PFirst` is a value from the domain of `person`.

More complex identifiers are often built upon these simple ones in combination with the checks of word forms or lemmas, as can be shown by the following example:

```
/* pronominal word form of the word "se" */

parse PositionType;

shared predex PronounSe;

PronounSe = ( (lower form == "se") and Pronoun ) ;
```

Similarly, `PronounSi` is defined (the word form `"se"` is replaced with `"si"` and `PronounSeSi` is based on these two identifiers as follows:

```
parse PositionType;

shared predex PronounSeSi;

PronounSeSi = (PronounSe or PronounSi);
```

Larger lists of word forms or lemmas should be stored rather in the vocabularies (see Section 5.3.4) than directly in the predicate expression variables.

The compiler uses a simple algorithm that looks for the definitions of the identifiers (including rules and groups), hence it is necessary to put the definitions into the files with the same name as the identifier itself, however, converted into the lowercase. For example, the two definitions of `PronounSe` and `PronounSeSi` should be put in the files `pronounse.rlm` and `pronounsesi.rlm` and the search path must be set properly, see [2] or Chapter 6.

5.3.13 Formatting comments

The rules written in *LanGR* for the disambiguation of Czech are all stored in the web-oriented database, see Appendix A. One of the advantages of the web presentation of the rules is that the content of the rules can be seen highlighted. While most of the rules' content is highlighted automatically (in dependence of the meaning of the identifiers), the formatting of comments remains in the user's competence.

The full list of comment-formatting commands can be found in the manual of *LanGR*, see [2], however, the following simple example should make most of the commands clear. It is a comment taken from the real rule `SlovesoSesSis`. The

comment is taken almost untouched from the bottom of the rule and describes the whole operation of the rule:

```
   /*
Pravidlo pro tvary "ses" a "sis": je-li v téže klauzi
slovesný tvar, pak je to minulé či pasivní příčestí,
infinitiv, přechodník nebo tvar "by", ale žádný jiný
slovesný tvar (mimo parentetické slovesné tvary).
Navíc musí být minulé, pasivní příčestí a přechodník
v singuláru, a pokud je u nich vyjádřena osoba, musí
to být osoba druhá.
\par
Pravidlo má tři slovosledné varianty z hlediska vzájemného
slovosledného postavení tvaru "ses"/"sis" a slovesného tvaru.
Slovosledná varianta \it{slovesný tvar ... "ses"/"sis"}
se rozpadá na dvě podvarianty:
\itemize
\item
ta prvá je obdobou varianty
\it{"ses"/"sis" ... slovesný tvar},
\item
druhá zachycuje případ, kdy slovesným tvarem je minulé
příčestí, a tehdy smí mezi tímto minulým příčestím a
tvarem "ses"/"sis" stát jen tyto tvary:
\it{"ty", "by", "už", "již"}.
\enditemize

\appliesto{Petře, kdy sis ty své vlastní materiály odnesl?}
Zde pravidlo odstraní prézentní slovesnou interpretaci
tvaru \bf{vlastní}.

\doesnotapplyto{Vytvořila by sis dobrou výchozí pozici}.
Tato věta je ovšem zcela v pořádku a musí se připustit,
čili posloupnost \it{"by ses/sis"} pravidlo musí akceptovat.
*/
```

In the comment, the several special commands are presented. A command always starts with a backslash followed by a command name (e.g. `\item`). Optionally there can be an argument passed to the command — it is then enclosed in curly brackets (e.g. `\bf{ses}`)[12].

The following comment-formatting commands are shown in the example above:

`\it{arg}` the argument is printed in italics;

`\bf{arg}` the argument is printed in bold;

`\par` introduces a new paragraph of the text;

`\nl` introduces a new line;

`\appliesto{arg}` the argument is the sentence the rule applies to; `arg` must not contain any further formatting, since it is planned that such sentences would be used in the automatic checking of the rule correctness;

[12] Obviously, this syntax was borrowed from the typesetting system TEX.

`\doesnotapplyto{arg}` `arg` is the sentence the rule does not apply to — opposite to `\appliesto`.

A couple of other commands is also available (e.g. color specifications), see [2]. Unrecognized commands are left untouched. The sample comment given above would appear in the web-presenter as follows:

> /* Pravidlo pro tvary "ses" a "sis": je-li v téže klauzi slovesný tvar, pak je to minulé či pasivní příčestí, infinitiv, přechodník a tvar "by", ale žádný jiný slovesný tvar (mimo parentetické slovesné tvary). Navíc musí být minulé, pasivní příčestí a přechodník v singuláru, a pokud je u nich vyjádřena osoba, musí to být osoba druhá.
>
> Pravidlo má tři slovosledné varianty z hlediska vzájemného slovosledného postavení tvaru "ses"/"sis" a slovesného tvaru. Slovosledná varianta *slovesný tvar ... "ses"/"sis"* se rozpadá na dvě podvarianty:
>
> - ta prvá je obdobou varianty *"ses"/"sis" ... slovesný tvar*,
> - druhá zachycuje případ, kdy slovesným tvarem je minulé příčestí, a tehdy smí mezi tímto minulým příčestím a tvarem "ses"/"sis" stát jen tyto tvary: *"ty", "by", "už", "již"*.
>
> **Applies to:** `Petře, kdy sis ty své vlastní materiály odnesl?` Zde pravidlo odstraní prézentní slovesnou interpretaci tvaru **vlastní**.
>
> **Doesn't apply to:** `Vytvořila by sis dobrou výchozí pozici`. Tato věta je ovšem zcela v pořádku a musí se připustit, čili posloupnost *"by ses/sis"* pravidlo musí akceptovat. */

Of course, the particular comment formatting depends on the particular web-browser used for accessing the web-presenter.

5.3.14 More complex rules

It has been already said that the expressive power of *LanGR* makes it possible to implement any (algorithmically defined) rule. However, some practice with the core of *LanGR* is required to write the rules outside the framework of this tutorial section. Or at least there must be somebody with the knowledge of *LanGR* core that will supply necessary functions (written in the core) into the system.

The list of the most useful functions available for linguists so far can be found in Appendix B. The *LanGR* core is defined in [2].

5.4 *LanGR* and surface grammatical rules

The primary objective of this section is to investigate relations between formal definition of disambiguation rules (see Section 5.1) and the rules written in the *LanGR* formalism. To understand this technical section, it is useful to be familiar with the definitions of application functions in *LanGR* (`ITEM`, `SEQUENCE`, `UNIFY`, ...), but it is not essential.

5.4.1 Rule variants as disambiguation rules

In the theoretical approach to the disambiguation rules, it was useful to have match proposals on input of the rules, i.e. the most difficult part of the operation

of rules — looking for the match — is realized outside the rules. The match searching task is left for the disambiguation process (that, however, blindly tries all match proposals for all rules).

In practice, it is not necessary for the disambiguation process to combine all match proposals with all disambiguation rules — it is more efficient to let a disambiguation rule look for matches by itself (because every rule knows its configuration). Hence the input of a disambiguation rule is only a sentence (without a match proposal).

While checking whether a match proposal is a match in a sentence is a deterministic *problem* (i.e. a task with outputs *yes* or *no*) that can be solved by a finite-state machine, looking for a match in an input sentence is a non-deterministic *task* that must be solved by a non-deterministic finite-state *transducer* (the output of the transducer, of course, contains the particular match). The transducer cannot be deterministic because it has to look for all possible matches of the rule in a sentence. Hence, in practice, a disambiguation rule is a "serial combination" of a non-deterministic finite-state transducer (that looks for a match) and a deterministic executive (linear Turing) transducer (this is in accordance with the theory).

A disambiguation ordering must then be changed from the set $D \times P$ (rules $\times$ proposals) to $D \times B$ (roughly speaking: rules $\times$ branches of operation of the non-deterministic finite-state transducer). This change doesn't violate the theory of the disambiguation rules (the results of such a "practically modified" disambiguation process are identical to the theoretical ones), but branches of operation of a non-deterministic finite-state transducer are more difficult to use in the theory.

In *LanGR*, a disambiguation ordering is, in fact, defined only on the set of rules D and it is expected that every rule processes all of its branches before it stops and returns an output. The branches of operation are created by the `split` command (see Section 5.2.4) and the interpreter uses the backtracking algorithm to process all the branches — in the backtracking, the order of branches is defined by the order of command sequences in `split`.

This largely reduces the total number of possible disambiguation orderings (there is now only exactly $|D|!$ different disambiguation orderings). Moreover, the order of disambiguation rules is defined by their order in the groups which finally implies that *LanGR* uses always one (precisely defined) disambiguation ordering.

This fact can cause problems for order-dependent sets of rules, however, it would be probably impossible to process all disambiguation orderings in any practical case. Hence it is left for the internal policy of the linguists to write order-independent rules — see Section 5.4.2.

Now, let us show that every rule variant in *LanGR* is really a disambiguation rule.

It is necessary to show for every rule variant v in *LanGR* that v has two separated parts — the configuration one and executive one. The configuration part is in fact a match searching transducer[13]. The executive part is a deter-

[13]A match searching transducer can be defined by an algorithm similar to the one in Defi-

ministic Turing transducer that operates only on the matching subsequence and it outputs a partial disambiguation of the input sentence.

Every variant can be split into two parts by the commands they contain: the *configuration* part contains only the `ITEM` and `SEQUENCE` commands (and their modifications). The *executive* part follows the last `ITEM` or `SEQUENCE` in the variant.

The ambiguity of the program execution is implemented in the *LanGR* core by the `split` command. The `split` command is present only in the application functions `SEQUENCE` and `ITEM`. Hence the executive part of the variant is deterministic.

During its operation, the executive part of the variant always refers to the sentence via *variables* that are set in the configuration part (by assignment of `ITEM` or `SEQUENCE`). Hence the variant can never see positions outside the match.

So far, there are only three commands used in the executive part such that they modify the sentence — `UNIFY`, `DELETE` and `LEAVE`. All of them only delete tags, hence every variant outputs a partial disambiguation of the input.

It remains to show that every executive part uses only linear space during its operation. This is hard to prove without the detailed knowledge of the *LanGR* core, but the most complex command used so far is `UNIFY` (with all of its modifications), which sequentially collects all possible combinations of morphological features first and in the second step it deletes anything that does not match these combinations, i.e. it operates in the linear space. Roughly speaking, linear space will meet the requirements of the linguists unless the particular readings of the sentence are investigated (then the space-consumption will raise to exponential).

The configuration part consists of the `ITEM` and `SEQUENCE` commands. The `ITEM` command corresponds to a static component of the configuration and the `SEQUENCE` command corresponds to a stretching component of the configuration. The sets of tags that are accepted by the components are defined by the arguments of the commands (the first-order formulae). Thus a configuration transducer can be built from a configuration part of any variant.

None of the `ITEM` and `SEQUENCE` commands modifies the sentence. The positions inside the match are passed to the executive part by the variables assigned to the output of `ITEM` and `SEQUENCE`.

Neither `ITEM` nor `SEQUENCE` matches an empty position. Moreover, none of the commands matches an unknown word (a tag that starts with `X` in the MAJH tagset), i.e. no unknown word can appear in a matching subsequence.

From the facts given above it follows (without any formal proof) that every rule variant is a disambiguation rule (with an ordering defined by the `split` commands used in the stretching components of the configuration).

---

nition 5.5, see page 42.

5.4.2 Properties of rule variants

This section investigates relations between formal sub-classes of disambiguation rules (see the definitions in Sections 5.1.3 and 5.1.4) and the "types" of rule-variants written in *LanGR* formalism.

Let us start with a simple rule from Section 5.3.1:

```
rule TwoVerbs {
     rulevariant VarI {
        ITEM IsSafe FiniteVerb;                    // (1)
        ToDelete = ITEM Possible FiniteVerb;       // (2)
        DELETE FiniteVerb FROM ToDelete;
     };
};
```

The variant `TwoVerbs.VarI` is *safely based* (see Definition 5.18 on page 48) and it is also a *grammar checker* (see Definition 5.14 on page 47).

Whenever there is a sentence with two subsequent *safe* finite verbs passed to the variant `TwoVerbs.VarI`, the variant deletes all tags from the second verb — i.e. `TwoVerbs.VarI` is a grammar checker.

Now, let S be a sentence that can be disambiguated by the `TwoVerbs.VarI` variant, i.e. S contains two subsequent positions p_1, p_2 such that the first position (p_1) matches the line `(1)` of the variant and the second position (p_2) matches the line `(2)` of the variant. Let R be a set of tags deleted by `TwoVerbs.VarI` from p_2 and let S' be a partial disambiguation of S such that none of the positions p_1, p_2 in S' is empty. Hence R contains only finite-verb tags.

If tag sets on positions p_1, p_2 in S' and S are identical then the variant operates on S' and S in the same way (on positions p_1, p_2) and R is deleted from S'. If the position p_1 in S' is partially disambiguated in S' (in comparison with S), the line `(1)` still matches p_1. If the position p_2 in S' still contains at least one finite-verb tag then p_2 matches the line `(2)` of the variant and all finite-verb tags are deleted from p_2. Hence `TwoVerbs.VarI` leaves no finite-verb tag in S' and `TwoVerbs.VarI` is safely based.

With some exceptions, a rule-variant written in *LanGR* is a *safely based grammar checker* whenever it uses no conditional command in its executive part. In particular, it must use neither the direct `if` command nor the conditional unification (`UNIFY CONDITIONALLY`) — the optionality usually implies that the rule does nothing if the sentence is incorrect and such a rule is not (very probably) safely based.

A very important question is how the set of rules must look in order to be order-independent. If there exists a "dangerous" rule (i.e. a rule with an optionality in its executive part) in the set, the order-independence of the set is not implied by the theory of disambiguation rules.

Chapter 6

Implementation

This chapter focuses on the execution of the rules written in *LanGR* on a real corpus. Up to now, there are three ways of execution of the rules:

- using the *interpreter of LanGR*,
- compiling the rules using *fast LanGR*,
- converting rules into negative n-grams.

All approaches have their advantages and disadvantages. In the interpretation of the rules (see Section 6.1), it is possible to execute everything written in *LanGR* (including the rules beyond the framework of the negative n-grams), however, without any optimization the interpreter is too slow to be used on really large corpora.

Fast LanGR can be used to process corpora very quickly, although, in comparison with the interpreter, *fast LanGR* is not able to execute all the rules that can be written in *LanGR* (see Section 6.2).

Converting rules into a FSM brings relatively fast application of the FSM to a corpus, however, a lot of rules cannot be converted to a FSM and this conversion is not further investigated in the thesis.

6.1 The interpreter of the *LanGR* formalism

The rules can be executed by the so-called *interpreter* of *LanGR*. In fact, the source code is not really interpreted (in the sense of interpreted programming languages), but it is compiled into a binary executable file that can be then applied to an input text (corpus).

At the time the work on the formalism has started, it was obvious that the expressive power of the formalism must be equal to the power of any conventional programming language. However, several special aspects of the rules' implementation appeared to be too complicated to be encoded directly in any existing programming language. It was decided to develop a completely new programming language (the *LanGR* core) with its own compiler.

Section 6.1.1 contains basic hints on how to get a text processed by the rules from the definition (source files) of rules, i.e. how to compile and execute a source code written in *LanGR*.

The interpreter of *LanGR* uses its own data format, which is defined on the application level of the formalism. Additionally, the input text of the rules must be processed by the E-MAJH analyzer (see Section 1.1). These issues are presented in Section 6.1.2.

6.1.1 Using the interpreter

This section is completely technical — it contains in fact an "algorithm" that describes how to process a source code of the rules to be applied to a text (corpus). Both compilation of the rules into binary executable and running the rules on a text is fully described in [2], however, a brief summary of all necessary steps is given here.

Since *LanGR* has been tested only on UNIX systems, particularly on Linux (see [25]), all operating system dependent operations presented in this section are demonstrated on this system and it is assumed that the reader is familiar with Linux at least on the user level. In any case, since the *LanGR* source code uses only standard C++ libraries, it is easily portable to other platforms.

First of all, assume that there exist some rules and other linguistic definitions (i.e. the source code of the rules that should be applied to a corpus) in one directory, e.g. `/rules` (it can be placed in any number of directories, but one directory gives more simplicity to what follows). Hence there exists a couple of source files with the suffix `.rlm` (which is a standard suffix of the *LanGR* files) in the directory `/rules`. Among all the files, let `root.rlm` be the file that defines the group `root` (see Section 5.3.11).

Then assume that *LanGR* compiler has been successfully installed on the particular machine, i.e. `make install` successfully passed (see installation hints in the distribution; the particular installation directory of *LanGR* is significant, see below). Now, the only important thing is the path to the compiler's executable — let `/bin/compiler.x` be this path.

Now it is possible to compile the source rules into one big C++ file:

```
$ /bin/compiler.x -e -o unix \
> -p /rules/ -i root -f /tmp/compiled.C
```

The complete list of switches accepted by the compiler can be found in [2] or in the on-line help of the compiler (accessible, for example, by executing the compiler with no arguments). Here only the most frequent switches are shortly explained:

`-e` stands for *compile*, it selects the action that compiler has to do;

`-o unix` selects the output character set (`unix` is a nickname for `ISO-8859-2` here[1]), i.e. the character set used in the output file;

[1] Only a few character sets are accepted so far, but it is not a big deal to add new character sets.

`-p /rules/` defines the path where source files are stored (final slash is mandatory, `-p` may be used multiple times);

`-i root` defines the module (source file) that should be processed (in this case, it is the `root` module, i.e. the file `root.rlm` will be processed); the definition files of unknown identifiers (e.g. members of the group `root`) are loaded automatically during the parsing, see [2];

`-f /tmp/compiled.C` output file name; the rules are converted into a C++ file of this name; write permissions are required to create the file.

If no error appears during the compilation, the output file (in this particular case, `/tmp/compiled.C`) is generated. If the compiler encounters any errors, error message is written to the standard error stream (the terminal, usually).

Now, this C++ file has to be compiled by a usual C++ compiler (e.g. `g++`, see [44]) into a binary executable. It is necessary to remember the installation paths of the compiler (see the variable `INSTALL_COMP_DIR` which contains the installation path of headers and libraries that will be required by the C++ compiler). Let `INSTALL_COMP_DIR=/lib/rules` be the definition in the `Makefile` of the compiler. Then the following command creates the executable:

```
$ g++ -o /bin/rules.x -I/lib/rules -I- \
> -L/lib/rules /tmp/compiled.C -lrl -lstdc++
```

Description of `g++` switches is far beyond this thesis, however, the executable rules should be stored in `/bin/rules.x` now.

The rules are prepared for their application to a text. However, *LanGR* executable defines its special input/output format, i.e. the input text must be converted into this format and the output is presented in this special format as well (i.e. it requires a conversion again).

The input/output format processed by the executable is, in fact, directly a sequence of entities of the data types available in *LanGR* and only the source rules (or functions on the application level of *LanGR*) define what is expected as input at run-time. By convention, the expected input is usually a sequence of instances of the data type `Text` that represents one sentence.

Hence it is necessary to convert any text that doesn't use *LanGR* internal format into this format — see Section 6.1.2. Let us assume that a file `/tmp/test.internal` contains an input text in the internal formal of the interpreter of *LanGR*. It can be processed by the rules `/bin/rules.x`:

```
$ /bin/rules.x -r root -i unix -o unix -e unix \
> -Rfirstoverload < /tmp/test.internal > /tmp/test.internal.out
```

Again, the full list of accepted switches can be found in [2] and in the on-line help of `/bin/rules.x` (accessible by `/bin/rules.x -h`). The switches given above are simple: `-r root` denotes the rule/group that should be executed in loops on the input corpus (see Section 5.4 and also [2]), `-i/-o/-e` denotes the character set used for input/output/error streams and `-Rfirstoverload` is a magic switch — usually nothing works without this switch.

If no error occurs, the result is written into `/tmp/test.internal.out`. The file can be converted from the internal format to other formats — see Section 6.1.2.

So, this is the "ideal" way of processing a corpus by the rules. The real problem of the execution of the rules using the interpreter is that the interpreter operates very slowly. This is a tribute which is paid for several necessary features of *LanGR* and for its generality. However, the speed of processing could be very probably increased under conditions which can be met in most of the rules (see Section 6.2).

6.1.2 The E-MAJH analyzer in the interpreter

This section presents the implementation of the E-MAJH analyzer (see Section 1.1) and the implementation of conversions to/from the internal data format used by the interpreter of *LanGR*.

Up to now, the interpreter has been applied to the PDT corpus (see Section 1.4) that uses the MAJH tagset. It is assumed that any input text has already been analyzed by the MAJH analyzer.

First of all, let us describe the conversion scripts and their usage. All the conversions required by the interpreter are implemented in the `Perl` programming language (see [41]).

The convertor `pdt2internal.pl` performs the conversion from the output of the MAJH analyzer to the input of the rules — it reads the output of the MAJH analyzer, implements the extension from the MAJH analyzer to the E-MAJH analyzer and returns the text in *LanGR* internal format.

The convertor `internal2other.pl` reads an output of the *LanGR* interpreter and converts it into other formats (either the MAJH format used by the PDT corpus or a web-page that can be opened in any web-browser). Both scripts are included in the distribution of *LanGR* interpreter.

Both scripts are very simple to use:

- Let `/corpus/test` be a text analyzed by the MAJH analyzer. The following command converts the text `/corpus/test` into the internal format that can be read by the interpreter. The output is written into the file `/tmp/test.internal`:

  ```
  $ ./pdt2internal < /corpus/test > /tmp/test.internal
  ```

- Let `/tmp/test.internal.out` be an output of the interpreter, i.e. the file `/tmp/test.internal.out` is a text analyzed by the rules. The output of the rules can be converted using the script `internal2other.pl` into two formats:

  - The following command converts the file `/tmp/test.internal.out` into the MAJH format:

    ```
    $ ./internal2other.pl csts /corpus/test.comments \
    > < /tmp/test.internal.out > /corpus/test.rules.csts
    ```

The result is written into `/corpus/test.rules`. The second argument (`/corpus/test.comments` in this case) of the script (if specified) is a file name that contains messages from the `REPORT` commands.

– The following command converts the file `/tmp/test.internal.out` into a web-page that can be opened in any web-browser:

```
$ ./internal2other.pl www < /tmp/test.internal.out \
> > /corpus/test.rules.html
```

This convertor is used in the WWW interface to the rules — see Appendix A.3.

Technical note: The scripts `internal2other.pl` and `pdt2internal.pl` include additional Perl modules — `PDT.pm` and `InternalFormat.pm`. The modules must be put in an appropriate directory (e.g. in the working directory) to be able to run the conversion scripts successfully.

The rest of this section is concerned with a few interesting technical issues of the conversion. It is not an exhaustive manual of how to write a converter to/from *LanGR* internal format, however, it can help a lot.

In the invocation of the compiled rules (see the Section 6.1.1) it is assumed that the rules are applied to a variable `Text` (i.e. a sentence) of the data type `TextType` (which defines what "sentence" means for the rules). The definition of the data type `TextType` directly affects the internal format of *LanGR*. It is because a constant of this type is what the executable rules expect on the input.

The format of the constant is identical to the constants in the source files of *LanGR* with some restrictions, see [2]. Structure constants are enclosed by {}, set constants are enclosed by `[]`. Elements of structures/sets are separated by a comma. Strings are enclosed in double-quotes. Predicate expressions (if any) on input must be specified in the bracket-notation.

So the input must precisely correspond to the data type `TextType` and this data type is mostly a non-trivial data type (which can be expected; usually, `TextType` is a structure data type).

For example, the current definition of `TextType` is

```
type TextType structure {
   string TextIdentifier;
   DeletionTrace TextReports;
   PositionTypeSet TextPositions;
};
```

Here `TextIdentifier` is an identifier of the sentence (a string), `TextReports` is a database of reports that are related to the whole sentence (a set type `DeletionTrace`) and `TextPositions` is a set of positions of the sentence (the set data type `PositionTypeSet`). Despite `TextIdentifier`, all other fields are structured again. The expected input of the compiled rules could look like

```
{"sent1",[],[<position1>,...,<positionn>]}
```

where `"sent1"` is a sentence identifier, `[]` is an empty set of reports and `<positions1>`, ..., `<positionn>` are particular positions of the sentence (currently of the data type `PositionType`). This input is loaded into the variable `Text` before the rules are executed. Whenever incorrect data type appears on input (i.e. a constant of a data type that does not correspond to the expected one), a run-time error appears and the execution stops.

6.2 *Fast LanGR*

This section shortly describes fast implementation of the *LanGR* formalism. The so-called *fast LanGR* implementation is fast enough to process large corpora by a lot of rules, but there are a few restrictions that make it impossible to compile an arbitrary rule (written in *LanGR*) using the fast *LanGR* implementation.

The fast *LanGR* implementation is based on the formal definition of a disambiguation rule — see Definition 5.7 on page 43. A rule can be encoded using fast *LanGR* only if it can be split into two separate parts: the *configuration* one and the *executive* one. The configuration part of the rule must contain the commands `ITEM` and `SEQUENCE` (and their modifications) only. In other words, it must be possible to encode the configuration part into a FST — up to now, all rules use only `ITEM` and `SEQUENCE` commands in their configuration parts.

The executive parts of the rules represent a more complex problem for the fast *LanGR* implementation. There is a wide variety of commands used in the executional parts, including conditional commands and the `UNIFY` command with its variants. There is no easy way of modifying all executional parts to contain more simple commands only (e.g. the `DELETE` command).

Both configuration and executive parts, however, use first-order formulae (see Section 5.2.3) to express the properties of tags and positions. Section 6.2.1 investigates the encoding of the first-order formulae in fast *LanGR*. The formulae represent basic elements that must be correctly put into a *skeleton* of a rule — i.e. into a FST that represents the configuration part of the rule and into the sequence of commands in the executive part. Up to now, the encoding of *skeletons* of rules is performed manually — Section 6.2.2 summarizes the issues related to this manual work and the ways toward a fully automatic encoding of rules.

6.2.1 Compiling first-order formulae

Fast implementation of first-order formulae (FOFs) is the primary objective that must be accomplished to achieve fast operation of the rules. This section describes automatic encoding of FOFs into "conditional expressions" that are used by the *fast LanGR* implementation.

At compile time, every rule variant is processed by the compiler to find all FOFs that are present in the rule — up to now, the list of commands that can contain a FOF is not very long: `ITEM`, `SEQUENCE`, `SELECT`, `UNIFY`, `UNIFIABLE`, `HOLDS`. Let's proceed with an example:

```
finv = ITEM IsSafe FiniteVerb
         and not (lemma member of AbyKdybyLemmas);
```

In this command, the following FOF can be found:

```
IsSafe FiniteVerb and not (lemma member of AbyKdybyLemmas);
```

Every FOF is compiled bottom-top. The compiler starts with the leaves of the syntactic tree of the formula and encodes them into appropriate "conditions". Then the compiler moves higher up in the tree and combines elementary "conditions" into more and more complex "conditions". Finally, the compiler adds the information about the quantifier (if any) into the "condition".

In the sample above, the syntactic tree of the command contains two leaves: `FiniteVerb` and `lemma member of AbyKdybyLemmas`. The `FiniteVerb` identifier is a FOF that defines a set of tags with the "finite verb" property. The set is generated by the compiler and stored in the tree of the command. The `lemma member of AbyKdybyLemmas` formula defines a set of lemmas that match the condition (i.e. those listed in the `AbyKdybyLemmas` vocabulary). The set of all lemmas that are refered to in all the rules is build up by the compiler and every `member` function defines a subset of this set. The set of lemmas that match the leaf is stored in the tree of the command.

Then the compiler proceeds with non-terminals — the `not` and `and` nodes. In this particular case, the `not` operator inverts the set stored in the `lemma member of AbyKdybyLemmas` leaf. Finally, the `and` operator combines the sets stored in the leaves into a single condition (in this case, the condition consists of two sets — one set of tags and one set of lemmas).

After the compiler finishes building up the condition "under" the quantifier `IsSafe`, it proceeds with adding information about the quantifier into the condition. In this particular case, the condition gets a flag that all the tags (with their lemmas) on a position must meet the condition to match the `finv` entry.

In practice, the positions are encoded as sets of tags (and lemmas) at runtime. Every set is encoded into its characteristic function (i.e. a bit set in C/C++), which makes it possible to verify whether a (partially disambiguated) position in an input text matches a condition very effectively.

It is beyond the scope of this thesis to describe the process of encoding FOF in detail, however, there are several issues that can be highlighted:

- the tagset must be fixed so that the encoding into fast *LanGR* were possible; any change in the tagset requires re-compilation of the rules;
- the set of lemmas that are referred to by the rules must be known at compile time; in practice, the set of lemmas is retrieved from the rules before the compilation of the rules begins;
- up to now, *LanGR* makes it possible for the linguists also to look at substrings (prefix, suffix) and lengths of lemmas — these formulae require special kind of conditions; in general, any other "type" of condition is a subject of a non-trivial programmer's work on fast *LanGR* (while the interpreter of *LanGR* is ready to implement a lot of new features very easily).

The input of the rules must be processed by the E-MAJH analyzer. In the interpreter of *LanGR*, the E-MAJH analyzer is encoded in the conversion script `pdt2internal.pl` (see Section 6.1.2). The fast *LanGR* implementation uses no conversion scripts — the E-MAJH analyzer is encoded directly in the run-time procedures. In particular, the `PDT.pm` module (that defines the conversion process between the MAJH and E-MAJH analyzers) is used to generate conversion table between the MAJH and E-MAJH tagsets. Hence there exists a relation that, for any tag from the MAJH tagset, assigns a set of tags in the E-MAJH tagset. The relation is generated at compile time. It immediately follows that the relation table must be re-generated whenever the MAJH tagset is changed.

In the end of this section, it must be said that the automatic compilation is available not only for the FOF, but also for another important phenomenon that is frequently present in the rules — the sets of categories used in the family of the `UNIFY` commands. To make the unification really fast, the `UNIFY` commands in fast *LanGR* use *unification classes*, i.e. the sets of tags where all tags within one set contain the same "key" values in the unification categories.

For example, if a `UNIFY` command performs unification only in the `case` category, there are seven unification classes available — one class for every value of `case`. The first class contains all nominative tags, the second class contains all genitive tags and so on.

The unification classes are generated at compile time together with the encoding of the FOF.

6.2.2 Encoding skeletons of the rules

Section 6.2.1 describes everything that can be (up to now) compiled automatically into the *fast LanGR*. Even though the automatic encoding of first-order formulae (and unification classes) into a really fast implementation is essential for the practical use of the rules, there is still a lot of work that must be done manually to run the rules.

Up to now, the *skeletons* of the rules are manually encoded in C++. Here the term *skeleton* refers to executive parts of the rules and to the transitions in finite-state transducers that represent configuration parts of the rules.

The manual encoding is not difficult, mainly because every rule variant is, in fact, already a standalone program. The disadvantages of manual encoding, however, are obvious: it can be a source of errors, it avoids the possibility of immediate debugging of new rules and it requires a team of programmers (the number of programmers is dependent on the number of linguists that write the rules). Last (but not least), manual encoding prevents potential new users from "playing" with *LanGR* — it is unrealistic to manually encode "hello world" rules written by *LanGR* beginners.

Finally, it is necessary to compile the complete rules automatically, however, it requires non-trivial changes to the compiler. The good new is that the compilation of first-order formulae seems to be a much more complicated task than the encoding of the skeletons (and, moreover, there is a lot of experience available from the manual encoding of the skeletons).

Chapter 7

Results

This chapter summarizes the tests that have been made by applying the rules on corpora. The chapter begins with two initial sections (the definition of quality measures (7.1) and the description of the test data (7.2)). Sections 7.3 and 7.4 investigate the input of the rules — the output of morphological analyzer and the ways of processing unknown words.

Section 7.5 shows the list of rules used in the tests. Section 7.6 investigates empty positions that can appear during the disambiguation process — the empty positions are extremely useful because they make it possible to identify the problems (errors) in corpora, in human annotation or in the rules themselves.

Sections 7.7 and 7.8 present the results of the rule-based disambiguation of corpora, i.e. the main objective of the disambiguation rules. Sections 7.9 and 7.10 concern less important (but also useful) applications of the rules — identifying errors in human annotation (or in the raw text) and verifying the output of stochastic taggers.

Section 7.11 briefly investigates other possibilities of the application of the rules in the natural language processing. Section 7.12 summarizes future work plans.

7.1 Measures of quality in the disambiguation task

The taggers (either stochastic or rule-based ones) are mostly evaluated by using the following three parameters — *precision* (p), *recall* (r) and *F-measure* (f).

Let C denote a sample corpus that has been processed by morphological analysis and later by a (partial) disambiguation algorithm. Let C^m denote the manual annotation of the corpus C. The manual annotation is assumed to be the "correct" one and it assigns always one tag to every position. Let

- m_C denote the number of running words (also referred to as tokens or positions) in the corpus C (including punctuation),
- h_C denote the number of positions i in C such that the tag t on position i in C^m is present in C in position i (i.e. h_C is the number of tokens with "correctly assigned tags"),

- t_C denote the total number of tags in the corpus C.

Then the characteristics of the disambiguation algorithm applied to C are defined as follows:

$$p = \frac{h_C}{t_C}$$
$$r = \frac{h_C}{m_C}$$
$$f = \frac{2pr}{p+r}$$

For full disambiguation algorithms (including most stochastic taggers), t_C is equal to m_C, hence $p = r$ and $f = p$. The measures become different for partial disambiguation algorithms (e.g. the rules written in *LanGR*).

It must be said that the measures can be used for checking the quality of analysis of anything (not only morphological disambiguation) without any change. Their linguistic inadequacy is currently criticized by more and more linguists. For example, precision and recall say nothing about the types of errors in the corpus C.

On the other hand, it is easy to obtain these numbers while any other quantitative information must be retrieved manually up to now. Hence precision, recall and F-measure will be used also to compare presented rules with other taggers.

All the characteristics are given in percentual ratios, i.e. multiplied by 100.

7.2 Test data

All the tests reported in this chapter are based on the Prague Dependency Treebank ([14]), version 1.0 (with some corrections that have been made on the way toward the version 2.0).

In the tests that require manual review of the results, so-called *devtest data* is used (morphologically annotated development test data). Whenever the results can be automatically retrieved, so-called *etest data* is used (morphologically annotated evaluation test data).

The *devtest* data contains 129,574 tokens in 8,244 sentences. Out of these sentences, 738 sentences are empty (no token in sentence), i.e. the devtest data contains **7,506** non-empty sentences. The *etest* data contains 124,957 tokens in 8,046 sentences. Out of these sentences, 815 sentences are empty, i.e. the etest data contains **7,231** non-empty sentences.

The data has been changed before its processing by the rules, because conditionals like *aby*, *kdyby* are split into two separate tokens in PDT — e.g. every *abych* is replaced by the token *aby* followed by a new token *bych*. Such a sequence *aby bych*, however, is not grammatical and it is refused by the rules. All occurrences of such a split token are amalgamated into the original single token (232 splits in *devtest*, 216 splits in *etest*).

Finally, the number of tokens in the tests presented decreased in comparison with the publicly available PDT distribution — the modified *devtest* corpus contains **129,342** tokens, modified *etest* corpus contains **124,741** tokens.

corpus	m	t	h	p	r	f	$\|T\|$
C_{d-pdt}	129342	482592	127092	26.335	98.260	41.537	4420
C^E_{d-pdt}	129342	2073567	127273	6.138	98.400	11.555	8130
C_{e-pdt}	124741	464081	122570	26.411	98.260	41.699	4420
C^E_{e-pdt}	124741	1753416	122749	7.001	98.403	13.072	8130

Table 7.1: The figures for morphologically annotated corpora

Both *devtest* and *etest* corpus contain manual morphological annotation and two morphological annotations performed by stochastic taggers:

- exponential tagger written by Jan Hajič, see [19, 20];
- HMM tagger written by Pavel Krbec, see [30].

7.3 Morphological analysis

The *devtest* and *etest* corpora have been processed by the MAJH and E-MAJH morphological analyzers (see Section 1.1). This section summarizes the characteristics of the *devtest* and *etest* corpora after the morphological analysis (which is exactly the input of the rules) — see Table 7.1.

A human annotator can assign a tag that is not available in the offer of the analyzer, hence the recall measure for a morphological analyzer may be less than 100 %.

The corpora C^E_{d-pdt} and C^E_{e-pdt} denote the *devtest* and *etest* corpora processed by the E-MAJH analyzer. The corpora C_{d-pdt} and C_{e-pdt} denote the *devtest* and *etest* corpora processed by the MAJH analyzer. The last column $|T|$ in the table denotes the size of tagsets of the individual analyzers. The size of the tagset increased in the E-MAJH tagset as compared to the MAJH tagset almost twice. The table also shows that the average ambiguity (i.e. number of tags per token) in MAJH is about 3.7 while the ambiguity in E-MAJH is about 16, i.e. the average ambiguity increased 4 times!

In the disambiguation task, any increase in the initial average ambiguity generally requires additional knowledge to be encoded in the disambiguation algorithm. Is it really necessary for the rules to increase the input average ambiguity so much? First, the ambiguity in E-MAJH increases mainly due to one-to-many replacements of tags — above all, every digital numeral (that is assigned a single tag in MAJH) is assigned tens of tags in E-MAJH. There are also other extreme examples — the word form *jehož* is assigned a single tag in MAJH and almost 60 tags in E-MAJH. These one-to-many replacements considerably increase the average ambiguity, of course. Another regular source of ambiguity increase is the *clsep* category that causes replacement of every single punctuation tag (in MAJH) by two tags (in E-MAJH).

The detailed study of the sources of ambiguity increase in E-MAJH is not presented in the thesis — it is a purely linguistic problem. It can be expected

that the E-MAJH algorithm will be updated in future to make it possible to write new types of rules.

Let *E-MAJH*(t) denote a set of all tags that are generated from a tag t (from the MAJH tagset) during the operation of the E-MAJH analyzer (i.e. *E-MAJH*(t) contains tags from the E-MAJH tagset).

The serious problem in C^E_{d-pdt} (the same holds for C^E_{e-pdt}) is that manual annotation is not available (manual annotation in PDT uses the MAJH tagset). The manual annotation of C^E_{d-pdt} was created from manual annotation of C_{d-pdt} as follows: If t is a manually assigned tag (from the MAJH tagset) on position p in C_{d-pdt} then the "manual" annotation of C^E_{d-pdt} on position p contains the set *E-MAJH*(t). It is clear that it can (frequently) happen that manual annotation of position in C^E_{d-pdt} contains more than one tag.

A tag t on position p of the morphological analysis of C^E_{d-pdt} is "correctly assigned" whenever t is present on position p of the manual annotation of C^E_{d-pdt}. As a consequence, a partial disambiguation of C^E_{d-pdt} is correct (on a position p) whenever the intersection of the set of tags that remain on p after the disambiguation and the set of tags on p in manual annotation of C^E_{d-pdt} is non-empty.

Naturally, precision considerably decreases in C^E_{d-pdt} (C^E_{e-pdt}) in comparison with C_{d-pdt} (C_{e-pdt}) — it is a direct consequence of the average ambiguity increase. It can be a surprise that the recall increases — it can happen because human annotators sometimes select a particular tag (i.e. a tag from the E-MAJH tagset, roughly speaking) instead of a tag-shortcut from the MAJH tagset.

7.4 Problems in input of the rules

This section investigates problems that can appear during the conversion of a corpus from its original state (raw text) to the input of the rules (output of *E-MAJH* analyzer). Basically, there are two types of problems related to processing prior to application of the rules: a problem related to a single token and a problem related to more than one token.

The group of more-than-one-token related problems contains errors in a segmenter (i.e. in a machine that identifies sentence boundaries) and errors in a tokenizer (i.e. a machine that defines the term *token*). If the segmenter incorrectly places sentence stop mark into the text then it can cause incorrect operation of the rules (e.g. if the segmenter joins two sentences that should be separated then the rules can realize that there are two finite verbs in a clause). Similarly (but less trivially), the tokenizer must cooperate with the rules — for example, if the rules expect that the sequence *je-li* is a single token then it can cause a problem when the tokenizer splits the sequence *je-li* into three tokens *je*, the hyphen (-) and *li*.

It seems to be simple to define what the term "error in segmenter" means. On the other hand, it is difficult to formally define a "token". For example, a linguist can expect that a date *26. června 1995* is a single token while some other linguist can expect that this date is split into four tokens *26*, the dot (.), *června*, *1995*. That's why it is difficult to say what the term "error in tokenizer"

means.

The *devtest* and *etest* corpora are already processed by a segmenter and a tokenizer. It is difficult to determine segmentation and tokenization errors in corpora and these errors cannot be automatically marked up in the input of the rules to prevent incorrect operation of the rules. On the other hand, "incorrect" operation of the rules may highlight segmentation and tokenization errors — see Sections 7.9 and 7.10.

The second type of problems in the input of the rules are problems related to a single token. Such a problem is produced by one of the filters in the pre-rule stage — whenever this happens, the problematic token is assigned an *invalid* flag. Technically, invalid tokens contain no tag in the input of the rules (i.e. after *E-MAJH* algorithm terminates). Invalid tokens cannot be taken into consideration by any rule during matching the rule's configuration, i.e. no match of any rule can contain an invalid token. On the other hand, it is allowed to find a match elsewhere in the sentence, so the disambiguation is not stopped at all.

A token can be assigned an *invalid* flag for one of the following reasons:

1. invalid for *technical reasons* — this concerns the tokens that somehow violate the requirements imposed by the implementation of the rules applied to the tokens and their morphological analysis. In the majority of cases, the number of lemmas returned by morphological analysis is too high[1];

2. *unknown* in the morphological analysis — the token is assigned an "unknown" tag by the *E-MAJH* analyzer;

3. part of a *collocation* — the token is part of a collocation. In the current implementation, the term "collocation" refers to "non-grammatical" collocation, i.e. a sequence of words that violates general grammatical rules of the language; the list of collocations currently contain only 136 entries (e.g. *dole bez*, *od nevidím do nevidím*).

The above list is top-bottom "exclusive", i.e. if a token is invalid for technical reasons, it can be neither unknown for morphological analysis, nor part of a collocation; if a token is unknown for morphological analysis, it cannot be part of a collocation any longer.

A token is *valid* if it has none of the above-mentioned properties. Valid tokens can be freely processed by the rules. Table 7.2 shows the number of tokens in the corpora used in the tests.

Now it remains to clarify the relation between validity of tokens and correctness of morphological analyzers w.r.t. the manual annotation (see Table 7.1). Table 7.1 shows comparison of the human annotation with the output of the morphological analyzers. As for the token types in Table 7.2, all token types (except for *invalid for tech. reasons*) are regularly processed by morphological analysis (both in *MAJH* and *E-MAJH* analyzers). The results can be compared with the appropriate manual annotation.

[1]The maximum number of lemmas in the *fast LanGR* implementation is set to 8.

token type	C^E_{d-pdt}		C^E_{e-pdt}	
all tokens	129342	100.000%	124741	100.000%
invalid for tech. reasons	105	0.081%	72	0.058%
unknown from morph. analysis	1169	0.904%	1262	1.012%
part of a collocation	67	0.052%	44	0.035%
valid	128001	98.963%	123363	98.895%

Table 7.2: Invalid tokens statistics

The tokens that have been put into the category *invalid for technical reasons*, however, represent a technical error that occurs during reading the output of morphological analysis (both *MAJH* and *E-MAJH*), i.e. such a token is assigned no tag by the analysis. To make it possible to compare these tokens with manual annotation (in Table 7.1), they were assigned an *unknown word tag* (in particular, `X@----------` in the *MAJH* tagset). It has already been said that the term "technical reasons" means almost always "too many lemmas" (on the output of morphological analysis). Thus, almost all tokens in the *invalid for technical reasons* category have been recognized by the morphological analyzer. Unlike the implementation of the *fast LanGR*, a human annotator was not limited by the number of lemmas in morphological analysis and it can be expected that he/she selected a regular tag (different from `X@----------`). Consequently, (probably) all the *invalid for technical reasons* tokens are counted as "errors" of morphological analyzers in Table 7.1.

It is difficult to conclude anything from the comparison of Tables 7.1 and 7.2. The manual annotation can contain "unknown word tags", so it cannot be said that the recall measure in Table 7.1 is somehow related to the number of tokens in the category *unknown from morphological analysis*. Table 7.2 only summarizes the reasons why some tokens make the operation of the rules impossible.

7.5 Knowledge of the language system

This section describes the set of rules used in the tests presented. The set of rules contains 1353 rule-variants in 277 rules. The rules have been downloaded on May 30th, 2005 from the web-interface, see Appendix A.2. The rules are continuously updated by linguists and it can be expected that their performance increases.

This thesis presents only a brief overview of the types (groups) of rules used in the tests — the full description of the particular rules can be found in the web-interface.

In the tests, the group `root` was executed, see Sections 5.3.11 and 6.1. The group `root` (as downloaded for the tests) contains the following sub-groups:

Jednorazova contains all the rules such that no other rule can help them in their work — usually these rules are based on the word forms only. The list of selected rules follows: *PropriaNezivSG* (it deletes plural in

proper names), *NeGenitivPluralFemJmen* (heuristic rule for genitive plural feminine of proper names), *BytAdjNeutCommaConj* (agreement of a verb, adjective and an interrogative pronoun), *SlovoSeCislo* (vocalization of *s/se* before digital numerals), *ShodaVKoordinaci1* (investigates trivial cases of agreement between predicate and coordinated subject);

Adjektiva contains two rules *SubstVerbaleAdj* (ambiguity of verbs and adjectives) and *SyntAdjektivum* (ambiguity of syntactic adjectives and nouns);

Castice removes particle tags whenever possible;

Cisla disambiguates written numerals;

HomonymieSe contains the rules *SeSiCislo*, *SeSloveso*, *SlovoSe*, *SlovoSe2* that resolve the ambiguity of the word *se*;

Klause specifies what is a clause separator;

Komparativ contains one rule *MnohemComp* that leaves only comparative degree after the word *mnohem*;

PadovaHomonymie — the sub-groups *Nominativ*, *Genitiv*, *Akuzativ*, *Vokativ* and *Lokal*; every group contains the rules that investigate the properties of the particular morphological case;

Predlozky contains the rules that resolve the ambiguity of prepositions;

Propria — *PropriaPersNames* associates the first name and a family name, *PanNovak* considers a pair composed of an appellative noun and a proper noun, *MestoPraha* investigates pairs of forms (lemmas) like *Město Praha* and *řeka Ob*;

Shoda performs disambiguation based on grammatical agreement, mainly an agreement in nominal phrases and a subject–predicate agreement;

Slovesa contains the rules that resolve the ambiguity between a verb and another part-of-speech tag(s);

Val_Adjektiv here disambiguation is based on the valency of adjectives;

VztazneVety contains the rules *Jehoz* (the rule distinguishes between possessive and non-possessive use of the lemma *jehož*), *JizUz* (it resolves the ambiguity of the word *již*), *TenKtery*, *NounKtery* (they both search for antecedents of an interrogative pronoun), *KteryAKtery* (coordination of subordinate clauses both initiated with *který*), *KteryVerbKtery* (two pronouns *který* separated by a verb), *BytAdjCommaKtery* (agreement of a verb, an adjective and an interrogative pronoun), *TyKtery* (pronoun *který* preceded by a personal pronoun);

Zajmena resolves the ambiguity of pronouns.

7.6 Empty positions

The primary objective of the disambiguation rules is to disambiguate corpora. It can happen during the disambiguation process that a valid token loses all of its tags — i.e. an *empty position* is produced *by the rules*. Every empty position (including invalid tokens, see Section 7.4) has two main properties:

- if an empty position occurs in the text, it is dangerous for the operation of the rules and the operation of the rules on such a position must be avoided;
- once a position becomes empty, it remains empty till the end of the disambiguation process — at the end of the process it is useful to study the source of "emptiness" of such a position.

From the implementational point of view, it is easy to ensure that an empty position never occurs in a match of any rule, so both invalid tokens and empty positions that occur during the disambiguation process are handled correctly.

Empty positions in the output of the disambiguation process mostly signal some grammatical error in the sentence. In a partial disambiguation of corpora, however, it is usually required that every token be assigned at least one tag, i.e. non-grammatical sentences must be annotated, too — in order to achieve this goal, the following approaches have been used:

- *naive empty position processing* — every empty position is assigned all the tags it was initially assigned by the morphological analyzer; i.e. after the process of disambiguation is finished, every empty position gets all the tags from morphological analysis;
- *careful empty position processing* — if a sentence contains an empty position, all the tokens in the sentence are assigned their complete morphological analysis; i.e. after the process of disambiguation is finished, every sentence that contains an empty position gets all the tags from morphological analysis in all tokens.

If it is assumed that every empty position occurrence is caused by non-grammaticality of the input text, it is natural to say that the rules are unable to process this non-grammatical sentence and a *careful* approach is adopted which returns the complete morphological analysis of the whole sentence.

The careful approach, however, causes a considerably lower performance of disambiguation (see Section 7.7). That's why also *naive* approach was tested — it adopts a (naive) idea that if an empty position occurs in the output, the rest of the sentence is disambiguated correctly. This doesn't hold in general, of course, but it can be successfully accepted mainly in the case of local rules (e.g. agreement in a prepositional phrase).

It has already been mentioned that empty positions can help in spotting grammatical errors in corpora. In particular, every empty position produced by the rules (i.e. in a correct token) signals a problem "somewhere", because a set of correct disambiguation rules can never produce an empty position on a correct input.

The list of most frequent sources of empty positions follows:

- *error in the raw text* — a spelling error or a grammatical error in the source text of the corpus;
- *error in morphological analysis* — the analyzer recognizes the word, but it assigns incorrect tag(s);
- *error in segmenter* — incorrectly recognized end of sentence;
- *error in tokenizer* — incorrectly recognized tokens in the sentence (see Section 7.4 for brief information about problems related to tokenization);
- *error in rules* — an incorrect rule (e.g. a rule that doesn't consider all grammatical sentences in the language) is present in the system that causes damage in the disambiguation and it directly or indirectly leads to an empty position(s).

It is useful to split some of the entries of the list above into more "sub-categories". The *errors in the raw text* category is split into two sub-entries: *foreign language* and *spelling/grammatical errors*. It can be a subject of further discussion whether foreign language parts should be annotated in some way to avoid processing of this text. From now on, the term "foreign language" denotes any piece of non-Czech text in the corpus (including names, song titles and so on).

The entry *error in rules* is also split into two sub-entries: *nominative of nomination* and *systematic error in a rule*. The phenomenon of the nominative of nomination (e.g. *město Praha*) is very complex and hard to recognize automatically. It is beyond the framework of this thesis to describe this phenomenon in detail. The phenomenon of nominative of nomination in some way destroys the structure of a sentence in Czech and can cause incorrect operation of rules (if the rules are not careful enough). So it is useful to separate errors made by the rules and caused by the presence of the nominative of nomination from other errors of rules.

It can sometimes happen that a rule disambiguates more than one position (token) in its executive part. In dependence of the actual situation in the sentence and of the rule's executive algorithm it can happen that a rule produces more than one empty position during the single execution of its executive part. As a consequence, the number of empty positions produced by the rules is not really interesting. An incorrect execution of the executive part of a rule should be investigated instead.

For example, let r be a rule that considers grammatical agreement of a preposition followed by a noun. In a naive case, the configuration of r consists of two items (safe preposition, safe noun) and the executive part performs the unification of the positions in morphological case. Trivially, if a safe preposition (e.g. *pro*) is followed by a safe noun (e.g. *hrazdě*) such that there is no agreement in case then the rule r deletes all the tags in both positions in one execution of the unification. Two positions get empty, but there is only "one reason" of their emptiness (their "disagreement" in morphological case).

all errors	138	100.000%
nominative of nomination	51	36.957%
error in a rule	38	27.536%
foreign language	20	14.493%
spelling/grammatical error	13	9.420%
morphological analysis error	12	8.696%
segmenter error	2	1.449%
tokenizer error	2	1.449%

Table 7.3: Empty positions statistics

In the *devtest* corpus, there were 150 positions deleted by the rules (i.e. out of all valid tokens). These positions were produced by **138** executive parts of the rules (i.e. some of the rules deleted more than one position in one step of the disambiguation algorithm). In the *etest* corpus, there were 134 positions deleted by **120** executive parts of the rules.

The positions deleted by incorrectly applied executive parts of the rules in *devtest* corpus have been manually processed to analyze the source of the problem. Table 7.3 shows the results of the analysis. It can be seen from Table 7.3 that most empty positions are produced by the errors in rules (if nominative of nomination is considered to be a rule error then the errors in rules cause about 64 % of all empty positions). On the other hand, the rules can be updated on the basis of these tests to avoid particular errors in future. Regardless of the high ratio of the rule errors, the rest of empty positions (cca 36 %) is caused by other problems — in the raw text, in the morphological analysis and so on — and the origin of every particular error can be analyzed and removed (either by fixing the raw text, the morphological analyzer,...).

Finally, analyzing the empty positions (i.e. those produced by the rules during the disambiguation process) can help to increase the quality of corpora (and the morphological analyzer as well).

7.7 Disambiguating with rules

The rules have been used to partially disambiguate the *devtest* and *etest* corpora (C^E_{d-pdt} and C^E_{e-pdt}, respectively). Section 7.6 investigated positions that lose all tags during the disambiguation process — the analysis of empty positions can help to improve the corpora quality. This section presents the results of disambiguation by means of the measures defined in Section 7.1.

Let ${}^nC^E_{d-dis}$ and ${}^nC^E_{e-dis}$ denote the corpora obtained by the disambiguation of C^E_{d-pdt} and C^E_{e-pdt}, respectively, after fixing the empty positions using the *naive* approach (see Section 7.6). Let ${}^cC^E_{d-dis}$ and ${}^cC^E_{e-dis}$ denote the corpora obtained by the disambiguation of C^E_{d-pdt} and C^E_{e-pdt}, respectively, after fixing the empty positions using the *careful* approach. Table 7.4 summarizes the results of the disambiguation. The column d in the table shows the ratio

corpus	m	t	h	p	r	f	d
C^E_{d-pdt}	129342	2073567	127273	6.138	98.400	11.555	N/A
${}^nC^E_{d-dis}$	129342	637012	127012	19.939	98.199	33.148	30.721
${}^cC^E_{d-dis}$	129342	963194	127070	13.193	98.243	23.262	46.451
C^E_{e-pdt}	124741	1753416	122749	7.001	98.403	13.072	N/A
${}^nC^E_{e-dis}$	124741	557519	122471	21.967	98.180	35.901	31.796
${}^cC^E_{e-dis}$	124741	802688	122528	15.265	98.226	26.424	45.779

Table 7.4: The figures for partially disambiguated corpora

corpus	average ambiguity
C^E_{d-pdt}	16.031
${}^nC^E_{d-dis}$	4.925
${}^cC^E_{d-dis}$	7.447
C^E_{e-pdt}	14.056
${}^nC^E_{e-dis}$	4.469
${}^cC^E_{e-dis}$	6.435

Table 7.5: The average ambiguity before and after disambiguation

$$d = \frac{\text{number of all tags after disambiguation}}{\text{number of all tags before disambiguation}} \tag{7.1}$$

Table 7.5 presents another useful ratio: the average ambiguity (number of tags per token) before and after disambiguation. It can be concluded from the tables that the rules are able both to considerably decrease the average ambiguity and simultaneously to keep the recall measure almost unchanged. To achieve full disambiguation, however, the rules are still too "weak". It is an open question whether it is possible to write a set of disambiguation rules that are able to achieve full disambiguation on every input sentence. It can be expected that the answer is *yes*, *but* more and more heuristic rules will be required.

Another open question is whether the way that leads to full disambiguation can be found in purely rule-based approach or whether the best disambiguation algorithm (for Czech, in particular) is a combination of stochastic and rule-based methods — such a successful combination of stochastic and rule-based disambiguation (with fully disambiguated output) was introduced [1].

From Tables 7.4 and 7.5 it can be concluded that the *naive* approach (of correcting empty positions) performs better than the *careful* approach — the recall measure is almost identical in both approaches, while the precision is considerably higher when the naive approach is used (and, as a consequence, average ambiguity is considerably lower in naively disambiguated corpora).

Based on the comparison of naive and careful approaches, it could be concluded that the rules usually don't "destroy" the whole sentence if an empty position appears. Such a conclusion, however, cannot be accepted as true without further investigation of the disambiguated data — such a detailed study is not part of this thesis.

corpus	m	t	h	p	r	f	d
C_{d-pdt}	129342	482592	127092	26.335	98.260	41.537	N/A
${}^{n}C_{d-dis}$	129342	242959	126808	52.193	98.041	68.121	50.345
${}^{c}C_{d-dis}$	129342	278179	126865	45.606	98.085	62.262	57.643
C_{e-pdt}	124741	464081	122570	26.411	98.260	41.699	N/A
${}^{n}C_{e-dis}$	124741	230133	122282	53.135	98.029	68.915	49.589
${}^{c}C_{e-dis}$	124741	265232	122339	46.125	98.074	62.742	57.152

Table 7.6: The figures for partially disambiguated corpora after reconversion to the *MAJH-tagset*

7.8 Converting back to the MAJH tagset

Section 7.7 presented partial disambiguation of *devtest* and *etest* corpora by disambiguation rules. The disambiguation rules, however, required serious changes of the morphological analysis (see Section 7.3). These changes make it impossible to compare the presented results with other published results of Czech taggers that use Hajič's morphological analyzer unchanged.

To make such a comparison possible, the disambiguated corpora have been "translated" back to the original MAJH-tagset. This section shortly describes the reverse translation.

Let p be a position (token) in a corpus. Let T_p denote the set of tags such that the token on position p is assigned the set T_p by the MAJH analyzer. Let E_p denote the set of tags such that the position p is assigned the set E_p by the E-MAJH analyzer. Let E'_p denote the set of tags that remain on the position p after the partial disambiguation of the corpus (in the E-MAJH tagset).

The set T'_p of tags (in the MAJH tagset) that remain on position p after partial disambiguation is defined by the following algorithm:

1. Let $T'_p = \emptyset$; for all tags $t \in T_p$ do the steps 2–3;
2. let C^t be the set of tags from the E-MAJH tagset such that these tags are produced by the conversion of the tag t from the MAJH tagset to the E-MAJH tagset;
3. if $C^t \cap E'_p \neq \emptyset$ then set $T'_p = T'_p \cup \{t\}$.

Let ${}^{n}C_{d-dis}$ and ${}^{c}C_{d-dis}$ denote the corpus created by "reverse-translation" of the corpus ${}^{n}C^{E}_{d-dis}$ and ${}^{c}C^{E}_{d-dis}$, respectively. Let ${}^{n}C_{e-dis}$ and ${}^{c}C_{e-dis}$ denote the corpus created by "reverse-translation" of the corpus ${}^{n}C^{E}_{e-dis}$ and ${}^{c}C^{E}_{e-dis}$, respectively. Table 7.6 summarizes the characteristics of the disambiguated corpora after the reconversion to the MAJH-tagset (see formula (7.1) for the definition of the d measure). Table 7.7 presents the average ambiguity in the corpora after the reconversion to the MAJH tagset. Both *naive* and *careful* approaches show very little decrease in recall (in comparison with the MAJH analyzer, i.e. the corpora C_{d-pdt} and C_{e-pdt}). The difference between *naive* and *careful* approaches in the recall characteristic is (analogously to Section 7.7) also

corpus	average ambiguity
C_{d-pdt}	3.731
${}^{n}C_{d-dis}$	1.878
${}^{c}C_{d-dis}$	2.151
C_{e-pdt}	3.720
${}^{n}C_{e-dis}$	1.845
${}^{c}C_{e-dis}$	2.126

Table 7.7: Average ambiguity before and after the disambiguation after the conversion back to the *MAJH-tagset*

unnoticeable, so it seems that the *naive* approach can be used to process the empty positions without any serious loss on recall.

The difference between *naive* and *careful* approaches in precision (and also in the average ambiguity), however, is not so significant (in comparison with Section 7.7).

It would be interesting to answer a question why the precision increases much more in the disambiguation using the E-MAJH tagset (in comparison with the disambiguation using the MAJH tagset). It would require a detailed study of numbers of deleted tags in particular positions in the disambiguated corpora (and also a study of particular tags that have been deleted) to answer the question properly. Such a study (that is beyond the objective of this thesis) could also answer some questions about linguistic adequacy of tag shortcuts in the MAJH tagset (see Section 7.3).

Without such a linguistically based detailed study of deleted tags, it can only be concluded that the rules are probably good in deleting tags that are produced by the conversion from the MAJH tagset to the E-MAJH tagset, but their performance decreases with tags that remain unchanged during the conversion from the MAJH tagset to the E-MAJH tagset.

7.9 Correcting manual annotation

The rules based on the knowledge of the entire language system can be used also to verify manual annotation of corpora (see [15, 16]). The efficiency of this technique was proved on the German corpus NEGRA (the corpus was verified by a simple set of negative n-grams; see [7, 13]).

The rules written for Czech can be used in search for errors in manual annotation of Czech corpora. The manual annotation (unambiguous) is disambiguated by the rules and an error is found whenever an empty position appears during disambiguation.

The pilot test was made on the *devtest* corpus — **311** empty positions were produced by the rules in the manual annotation. These 311 positions were deleted by **293** executional parts of the rules (see Section 7.6). Every empty position in manual annotation indicates that there is an error somewhere — either in the rules, or in the text, or in the manual annotation. The empty

error source	e	
error in annotation	93	31.741%
nominative of nomination	83	28.328%
error in rules	58	19.795%
spelling/grammatical error	39	13.311%
incorrect tokenization	11	3.754%
incorrect segmentation	5	1.706%
foreign language	4	1.365%
total	293	100.000%

Table 7.8: Error sources of empty positions in human annotation

positions have been manually analyzed and every operation of the executive part of a rule that produced (at least one) empty position was stored into one of the error-source categories — see Table 7.8 (the column e contains the number of executive parts producing an empty position). The description of the error-source categories can be found in Section 7.6. In comparison with Section 7.6, Table 7.8 contains new category *error in annotation* — it contains errors caused by an incorrectly manually assigned tag (while the correct one could have been chosen by the annotator).

Table 7.8 (in comparison with Section 7.6) does not contain the category *error in morphological analysis*. For the disambiguation of manual annotation (which doesn't use any morphological analyzer), this error category is irrelevant.

Note: In fact, the category *error in morphological analysis* could have been present also in Table 7.8 — it could have contained cases when the correct tag was not offered to the human annotator by the morphological analyzer. However, it is impossible to obtain the version of the morphological analyzer that was used at the time of corpora annotation. Hence every "incorrectly assigned tag" was put into the *error in annotation* category.

From Table 7.8 it follows that the errors in rules (including the nominative of nomination phenomenon) caused about 48 % out of all empty positions. The rest of empty positions (52 %) is caused either by an error in the text or by an error in the annotation. It can be concluded (similarly to Section 7.7) that the rules can help in identifying the errors in corpora.

Also, both from Table 7.8 and from Section 7.7 it follows that the phenomenon of nominative of nomination should be deeply investigated in future.

7.10 Correcting stochastic taggers

Several stochastic taggers were used to (fully) disambiguate Czech texts, see [1, 18, 19, 20]. Up to now, the best stochastic tagger for Czech is the HMM-tagger presented in [1]. It reaches the recall measure at 95.16 %.

The paper [1] also presents a serial combination of the HMM-tagger and a few disambiguation rules. The rules are applied *before* the HMM-tagger to reduce the ambiguity of its input. The rules used in the paper [1] have been

later rewritten in the *LanGR* formalism. The combination of the HMM-tagger and the rules has improved the performance of the tagger — it reaches the recall 95.38 %.

As well as any algorithm that performs full disambiguation, the stochastic taggers mentioned above have one important property — their recall measure is equal both to precision and F-measure. The F-measure characteristic is usually accepted as the "most appropriate" characteristic of the performance of a disambiguation algorithm. From this point of view, the performance of the rules still stay far beyond the taggers due to low precision (cf. Table 7.4 on page 97).

Nevertheless, more and more rules are written in the *LanGR* formalism and it can be expected that combining the rules actually available in *LanGR* with the HMM-tagger would lead to an improvement in tagging. It is also possible to use other ways of combining stochastic taggers with the rules and look for "new" methods to achieve full disambiguation of higher quality.

This thesis, however, doesn't focus on combinations of taggers and disambiguation rules in general. This section concerns a possibility of using disambiguation rules as "verifiers" of the output of stochastic taggers. Verifying the output of stochastic taggers by the rules has two advantages:

- on "theoretical" level, the sentences that were refused by the rules on the output of the tagger can be manually analyzed to determine the error source (see below). In case of n-gram based stochastic models (e.g. HMM models), the knowledge of error sources would probably not help to improve the tagger's performance. But in case of models with some linguistic parameters (e.g. exponential models with linguistic features), the knowledge of error sources can help in selecting the features to be put into the model.

- on "practical" level, the "verifier" can be directly combined with a tagger to improve the performance of the tagger: if the tagger can produce the so-called *n-best list* (i.e. it can output not only one disambiguation which is the best one, but it outputs n best disambiguations) then the rules can be used to prune the n-best list and pass the first correct (w.r.t. rules) disambiguation to the output.

The verification itself is performed in the same way as the verification of manual annotation (see Section 7.9) — the output of a tagger (unambiguous) is passed to the rules. An empty position produced by the rules indicates a presence of an error.

The *devtest* corpus (see Section 7.2) is annotated by two different taggers — Hajič's exponential tagger (TH) and Krbec's HMM tagger (TK).

Outputs of the taggers were disambiguated by the rules. In the output of the TH tagger, the rules produced 1671 empty positions by **1377** applications of executive parts of the rules. In the output of the TK tagger, the rules produced 1688 empty positions by **1394** applications of executive parts of the rules. Every particular application of an executive part of a rule has been manually analyzed and put into one of the error-source categories — the results are presented in

error source	TJH		TPK	
error in annotation	1194	86.710%	1199	86.011%
nominative of nomination	72	5.229%	72	5.165%
error in rules	43	3.123%	46	3.300%
spelling/grammatical error	41	2.977%	46	3.300%
incorrect tokenization	5	0.363%	6	0.430%
incorrect segmentation	5	0.363%	5	0.359%
foreign language	17	1.235%	20	1.435%
total	1377	100.000%	1394	100.000%

Table 7.9: Error sources of empty positions in the outputs of taggers

Table 7.9. It can be seen from Table 7.9 that the taggers perform almost identically when compared as to quality of their output from the point of view of the rules. Most errors were found in the annotation, i.e. an incorrect tag was assigned by a tagger. The table also confirms the necessity to analyze the the phenomenon of nominative of nomination.

The detailed analysis of the tagger errors (i.e. why an incorrect tag was assigned by the tagger) is beyond the framework of this thesis — it is rather a task for the authors of the taggers. In any case, it follows from Table 7.9 that the rules are able to detect errors made by the taggers with high precision.

7.11 Other applications of rules

A formalized grammar of a natural language can be, of course, used in any application that processes a particular language — either in the "research" area (e.g. building corpora, investigating properties of the language system) or in the "commercial" area (e.g. grammar checking, speech recognition).

The rules are currently being used in a pilot project of grammar checking, however, no results are presented because there is no "corpus of errors" available for Czech. The rules can help a lot in building such a corpus by marking up "suspicious" sentences, see Section 7.6.

For other language modeling applications it is probably necessary to combine the rules with stochastic methods used so far in this area because the part of the language system knowledge encoded in the rules is still too small. The rules need not to be combined with a standalone stochastic tagger — they can operate as part of a combined tagger (e.g. the rules can operate as features in maximum entropy models) or as supervisors in supervised learning (the rules can mark up errors of the tagger and the tagger can use this information to improve its performance).

7.12 Conclusions and future work

From the whole thesis it follows that a lot of new rules is required to achieve considerably better results.

The set of rules is completely in the competence of the linguists, however, the implementation must support the ideas of linguists — one of the latest requirements of the linguists is valency frames, for instance. This can improve the performance of the rules a lot, but it also requires careful implementation, because large valency vocabularies must be processed.

For a long time, the crucial problem has been the performance of the implementation of the rules. The interpreted language *LanGR* is very slow — it takes 2–3 minutes to process one sentence. Such a very low performance, of course, disables effective writing and debugging the rules. Only recently a "fast *LanGR*" implementation has been developed, however, the linguists must still debug the rules using the slow interpreter or *LanGR*. To make the fast *LanGR* available for linguists is the main short-time goal.

For the fast *LanGR*, however, it is necessary to abandon some of the advantages of the interpreter of *LanGR*. For example, both the tagset and the repertory of the functions used in the rules must be fixed. It can be expected, also, that fast *LanGR* will have problems in case of additional linguistic requirements (e.g. using valency frames in the rules).

It is necessary to investigate the rules themselves, because the way of a much more effective implementation leads through good theory. It seems, however, that the rules must be investigated from the linguistic point of view more than from the mathematical one. For example, the term of order-(in)dependency (which is one of the most important properties of every set of rules) would probably have to be defined using the language system.

It is an open question whether a configuration part of a rule must be nondeterministic. Put simply, is it necessary to assume that more matches cross one another in one sentence? If matches cannot cross one another then the configuration part can be defined as the *deterministic* finite-state transducer that highlights *all* matches in the sentence at once. This would increase the performance a lot.

A separate study can focus on the operation of rules for a non-grammatical input. How much do disambiguation rules "destroy" an incorrect sentence? In other words, what is the practical difference between the naive and careful processing of empty positions?

Another good question is whether an automatic grammar corrector can be developed on the basis of the rules. If S is a sentence such that a disambiguation process of S by a set of rules produces some empty positions, is it possible to *automatically* correct the error in the sentence? As a consequence, is there any basic difference between grammar correcting and grammar checking?

It is being heavily discussed now whether it is necessary to add information to the sentence — it means not only to add "syntactic" functions to tokens, but to add also relations between positions and even relations between single tags. The last point is, however, a step toward the use of the readings-notation which would move the formal definition of a disambiguation rule from *linear* Turing machine to the *general* Turing machine. With syntactic information added, full syntactic analysis could be performed by the rules (including optional constructions of syntactic trees).

In the set of rules currently maintained by the Czech linguists, the number

of rules that use "conditional" expressions in their executive parts increases — i.e. the rules are not safely based. Not only the whole set of rules can be order-dependent, but these rules play a "strange" role in grammar checking — such a conditional rule is not a grammar checker, but it can disambiguate and hence it can help some other rule to produce an empty position.

From Section 7.9 it follows that there is a lot of foreign titles and nominatives of nomination present in the PDT corpus (and probably also in other corpora), however, the occurrences of these phenomena are neither separated from the rest of the text, nor specially annotated. This is a challenge for corpora-maintainers to investigate the possibilities of annotating these phenomena — and the rules can help to produce such an annotation in existing corpora.

Acknowledgments

The thesis would probably not exist without the support of

- the *Institute of the Czech National Corpus* — supported by the *Czech Ministry of Education, Youth and Sports* (Grants No. VS 96 139, MSM 112100002),
- the *Institute of Formal and Applied Linguistics* — supported by the *Grant Agency of the Czech Republic* (Grants No. 405/03/0913, 405/96/K214) and the *Czech Ministry of Education, Youth and Sports* (Grant No. LN 00A063),
- the *Center for Computational Linguists* — supported by the *Czech Ministry of Education, Youth and Sports* (Grant No. LN 00A063)
- and the *Austrian Research Institute for Artificial Intelligence* (Austria) — supported by *Fonds zur Förderung der wissenschaftlichen Forschung* (Grant No. P12920).

The thesis would not be accomplished without the support of Vladimír Petkevič, Jan Hajič and Karel Oliva.

Special thanks to Petr Podveský for a lot of help related to the proof of Theorem 4.10.

Bibliography

Author's bibliography

[1] Hajič, J., Krbec, P., Oliva, K., Květoň, P., Petkevič, V.: *Serial Combination of Rules and Statistics: A Case Study in Czech Tagging.* In Proceedings of the ACL-2001, Toulouse, France, 2001

[2] Květoň, P.: *Language for Grammatical Rules*, technical report TR-2003-17 at IFAL MFF UK, Prague, 2003

[3] Jasoň et al.: *Automaty a gramatiky* (in Czech), `http://ktiml.mff.cuni.cz/downloads/Automstr_ps.zip`, 1995

[4] Květoň, P., Oliva, K.: *Achieving an Almost Correct PoS-Tagged Corpus.* In Proceedings of the Text, Speech and Dialogue 2002, Lecture Notes in Computer Science, vol. 2448, pp. 19–26, Springer-Verlag, Berlin, Heidelberg, 2002

[5] Oliva, K., Květoň, P., Ondruška, R.: *The Computational Complexity of Rule-Based Part-of-Speech Tagging.* In Proceedings of Text, Speech and Dialogue 2003, Lecture Notes in Computer Science, vol. 2807, pp. 82–89, Springer-Verlag, Berlin, Heidelberg, 2003

[6] Květoň, P.: *A maximum entropy approach to natural language modeling.* In Proceedings of contributed papers to WDS'99, MATFYZPRESS, Prague, 1999

[7] Květoň, P., Oliva, K.: *(Semi-)Automatic Detection of Errors in PoS-Tagged Corpora.* In Proceedings of the 19th International Conference on Computational Linguistics (COLING 2002), pp. 509–515, vol. 1, Morgan Kaufmann Publishers, San Francisco, 2002

[8] Oliva, K., Květoň, P.: *Linguistically Motivated Bigrams in Part-of-Speech Tagging of Language Corpora*, Prague Bulletin of Mathematical Linguistics 78, pp. 23–36, MFF UK, Prague, 2002

[9] Oliva, K., Květoň, P.: *(German) Corpus representativity, bigrams, and PoS-tagging quality*, KONVENS 2002, pp. x1–x8, DFKI, Saarbrücken, 2002

[10] Oliva, K., Květoň, P., Petkevič, V., Hnátková, M.: *The Linguistics Basis of a Rule-Based Tagger of Czech.* In Proceedings of the Conference on Text,

Speech and Dialogue 2000, Lecture Notes in Computer Science, vol. 1902, pp. 3–8, Springer-Verlag, Berlin, Heidelberg, 2000

Other references

[11] Karlsson, F., Voutilainen, A., Heikkilä, J., Antilla, A. (eds.): *Constraint Grammar — A Language-Independent System for Parsing Unrestricted Text*, Mouton de Gruyter, Berlin & New York, 1995

[12] Stroustrup, B.: *The C++ Programming Language*, Addison-Wesley, 1997

[13] NEGRA: A Syntactically Annotated Corpus of German Newspaper Texts, `http://www.coli.uni-sb.de/sfb378/negra-corpus`

[14] Linguistic Data Consortium: *The Prague Dependency Treebank (PDT)*, LDC2001T10, 2001

[15] Oliva K.: *The Possibilities of Automatic Detection/Correction of Errors in Tagged Corpora: a Pilot Study on a German Corpus.* In Proceedings of the Conference on Text, Speech and Dialogue 2001, Lecture Notes in Computer Science, vol. 2166, pp. 39–46, Springer, Berlin, Heidelberg, 2001

[16] Oliva K.: *Linguistics-Based PoS-tagging of Czech: Disambiguation of se as a Test Case.* In Kosta P. et al.: Investigations into Formal Slavic Linguistics. Contributions of the Fourth European Conference on Formal Description of Slavic Languages — FDSL IV held at Potsdam University, November 28–30, 2001. Peter Lang, Frankfurt am Main, Berlin, pp. 299–314.

[17] Oliva, K.: *A Parser for Czech Implemented in systems Q.* Explizite Beschreibung der Sprache und automatische Textbearbeitung, Vol 16, MFF UK, Prague, 1989

[18] Mírovský, J.: *Morfologické značkování textu: automatická disambiguace* (in Czech), Diplomová práce, MFF UK, Prague, 1998

[19] Hajič, J., Hladká, B.: *Tagging Inflective Languages: Prediction of Morphological Categories for a Rich Structured Tagset*, Proceedings of COLING-ACL Conference, Montreal, Canada, pp. 483–490, 1998

[20] Hajič, J.: *Morphological Tagging: Data vs. Dictionaries* Proceedings of 6th ANLP Conference / 1st NAACL Meeting, Seattle, Washington, pp. 94–101, 2000

[21] Hajič, J.: *Disambiguation of Rich Inflection (Computational Morphology of Czech)*, Karolinum, Charles University Press, 324 pp., 2004

[22] Hajič, J.: *Unification Morphology Grammar*, Ph.D. thesis at IFAL MFF UK, 1994

[23] Berger, Adam L., Della Pietra, Vincent J., Della Pietra, Stephen A.: *A Maximum Entropy Approach to Natural Language Processing*, Computational linguistics, vol. 22, no. 1, 1996

[24] Brill, E.: *A simple rule-based part-of-speech tagger.* In Proceedings of ANLP-92, 3rd Conference on Applied Natural Language Processing, Trento, IT, pp. 152–155, 1992

[25] *Linux Operating System*, `http://www.linux.org`

[26] *The Czech National Corpus*, `http://ucnk.ff.cuni.cz`

[27] *Standard Generalized Markup Language (SGML)*, ISO 8879:1986, `http://www.w3.org/TR/REC-html40/intro/sgmltut.html`

[28] Ginsburg, S., Rose, G. F.: *A characterization of machine representation of natural language.* Canadian Journal of Mathematics, 18., 1966

[29] Mohri, M.: *Finite-State Transducers in Language and Speech Processing*, Computation Linguistics vol. 23, 1997

[30] Krbec, P., Podveský, P., Hajič, J.: *Combination of a Hidden Tag Model and a Traditional N-gram Model: A Case Study in Czech Speech*, in EUROSPEECH 2003 Proceedings (8th European Conference on Speech Communication and Technology), vol. 3, pp. 2289–2291, ISCA, Geneva, 2003

[31] Hopcroft, J. E., Ullman, J. D.: *Introduction to Automata Theory, Languages and Computation.* Addison-Wesley, 1979

[32] Garey, M. R., Johnson, D. S.: *Computers and Intractability: A Guide to the Theory of NP-completeness.* W. H. Freeman, San Francisco, 1979

[33] *UNICODE*, `http://www.unicode.org`

[34] Pala, K., Rychlý, P., Smrž, P.: *DESAM — Annotated Corpus for Czech.* SOFSEM 1997, pp. 523–530

[35] Žáčková, E.: *Parciální syntaktická analýza (češtiny)* (in Czech). Dissertation thesis at the Faculty of Informatics, Masaryk University, Brno, 2002

[36] Ševeček, P.: *LEMMA — lemmatizátor pro češtinu* (in Czech). Brno 1996

[37] Doležalová, D.: *Počítačový modul stylistického korektoru češtiny* (in Czech). Master thesis at MFF UK, Prague, 2004

[38] Chomsky, N.: *Three models for the description of language.* IRE Transactions on Information Theory, 2 (1956), pp. 113–124

[39] *PHP scripting language*, `http://www.php.net`

[40] *MySQL*, `http://www.mysql.com`

[41] *Perl*, `http://www.perl.com`

[42] *Concurrent Versions System* `https://www.cvshome.org/`

[43] *Apache HTTP server*, `http://www.apache.org/`

[44] *GNU Compiler Collection*, `http://gcc.gnu.org`

[45] *GNU tar*, `http://www.gnu.org/software/tar/`

[46] Hrbacek, K., Jech, T.: *Introduction to Set Theory.* Marcel Dekker, Inc., New York, 1999

[47] Enderton, H. B.: *Elements of Set Theory.* Academic Press, New York, 1977.

Appendix A

WWW interface

The rules written in *LanGR* are available on the website

`https://slivka.ff.cuni.cz/loginform.php?`

The site is written in Czech. Also, the site restricts the access only to authorized users.

The site maintains the sources of rules and it enables the linguists to add new rules, change them and compile them into an executable. These features of the site, the so-called *stack* of rules, are described (on user-level) in Appendix A.2.

The site also allows the users to verify the operation of the rules. Authorized users can process their own sentences with the rules compiled in the *stack*. This feature of the site is described in Appendix A.3.

There is a mailing list available for the developers of the rules. The list is accessible on the website

`http://lists.czech-language.cz:372/wws`

and the e-mail address of the list is `rl@lists.czech-language.cz`. The mailing list is maintained by *SYMPA*, see `http://www.sympa.org/`.

A.1 Technical details

First of all, let's investigate several technical details of the website.

The website is written completely in the scripting language PHP ([39]). The stack of rules uses revision control system CVS and the database server MySQL ([40, 42]). The access to the site is encrypted by SSL (*Secure Socket Layer*), i.e. it uses SHTTP protocol instead of plain HTTP. The pages generated by the server are plain HTML pages, i.e. all the processing runs on the server side. The site is maintained by the *Apache* HTTP server, see [43].

The combination of the database system MySQL and the revision system CVS enables the server to remember the history of changes of every rule, group or application-level definition and simultaneously to search quickly through the identifiers.

The functionality of the site can be simply expanded due to its modular architecture (see Section A.2).

A.2 The storing of rules

First of all, the user that enters the web-interface has to authenticate himself by filling in the welcome dialog — the dialog itself already uses the SHTTP protocol, hence there is no worry about the disclosure of the passwords.

All the pages in the stack have the same structure, see the sample page on Figure A.1. At the top of a page, there is a *header* that contains the page's title and the *buttons* (HTML links to other pages) that represent the actions that are related to the current page.

The list of buttons is generated automatically in dependence of the page's content. Since nothing more can be found on the pages, the following list of available buttons describes the whole site.

The list of buttons is given in Czech. Although, since the buttons are generated automatically from a list, they can be simply translated into other languages.

Note: Only relevant buttons from the following full list are available on each page.

Button name	Action
Veřejné stránky	opens the window with publicly accessible site of *LanGR*
Chráněné stránky	in public part of the site, opens a login form to the private part of the site
Kontextová nápověda	opens a new window with contextual help for the actual page
Volby v záhlaví stránky	opens a new window with short description of the buttons in the header
Popis jazyka	opens the documentation of the core level of *LanGR*
Tutorial	opens the tutorial documentation (see Section 5.3 in this thesis)
Tabulka priorit	opens the table of priorities of built-in functions
Odhlášení	click to log off the stack of rules
Změna hesla	opens a page that makes it possible to change the password
Historie změn	not yet available
Tarball	creates an archive of all the source files actually present in the system (uses the archiver `tar`, see [45]);
Hledej	opens the search dialog — makes it possible to search in the names, descriptions and also the contents of the files
Spustit pravidla	opens a dialog that makes it possible to run the rules, see Appendix A.3

Přidat identifikátor	opens a form for adding a new identifier (rule, group or other); in the form, fill in the name of the identifier, short description and the file with its definition to be uploaded from your computer to the server
Globální identifikátory	opens the list of all global identifiers in the system — use the list of alphabet letters directly below the header to get a sub-list of the only identifiers starting with the particular letter
Skupiny	opens the list of all groups, see the button *Globální identifikátory* above
Pravidla	opens the list of all rules, see the button *Globální identifikátory* above
Kompilace	compiles the rules into an executable; it is finished either with a confirmation of successful compilation or with an error message; the executable is located on the server to be used in the testing page, see the button *Spustit pravidla* above
Obsah souboru	shows the raw content of the highest version of the file
Odstranit	removes the identifier and its file from the system; requires a confirmation
Přehled verzí	shows the list of all versions of the current identifier that have been committed to the system; useful for returning to older versions after making fatal errors
Nová verze	opens a form for adding a new version of the current identifier; just fill in the file name with the new version on your computer
Změnit název/popis	opens a form for changing the name and the description of the object
Odlišnosti od hlavní verze	shows the differences between the current version and the "head" version (i.e. the latest version); not implemented yet
Download	passes the plain file content to the browser
Syntax	performs syntactic analysis of the file and shows the file with highlighted tokens — most of the tokens are hypertext links to their definitions; an error message is produced on syntax errors

Figure A.1: Sample page from the stack of the rules

A.3 The testing of rules

The rules in the stack can be compiled into an executable which can be used to disambiguate sample sentences on the "testing pages". The testing pages can be entered by clicking on the *Spustit pravidla* button in the stack of rules, see Appendix A.2.

The HTML form that appears after clicking on the *Spustit pravidla* button is very simple to use. It makes it possible to process a single sentence (written directly into the text box in the form) or to upload a text file to be processed by the server.

The HTML form also makes it possible to select a character set you use on your computer (incorrect setting of this option can cause incorrect operation of the rules).

Once you submit your sentence/file to the server (by clicking on the *Veta* button or on the *Soubor* button), it is processed by the *interpreter* of *LanGR*[1]. As a consequence, it can take some time for the rules to process the text.

When uploading larger text files, it can be useful to fill in an e-mail address to the HTML form. The result of processing will be sent (as an attached HTML page) to the e-mail address instead of displaying it directly in the browser.

If a problem occurs during the rules' operation, an error message is displayed (or it is sent to the e-mail address). Otherwise, the list of analyzed sentences is opened (see Figure A.2). The sentences are numbered from 1. Every sentence has two buttons in front of it: *everything* and *non-deleted*. Clicking the *non-deleted* button opens a new window that shows the sentence after the partial disambiguation performed by the rules. The sentence is displayed on the top of the page. The list of "tags" that remain after the partial disambiguation is printed in a column below every position (see below a brief description of "tags").

The button *everything* in the list of analyzed sentences opens a new window that presents the sentence with complete morphological analysis (E-MAJH). The "tags" that remain after the partial disambiguation are printed in blue. The "tags" that are deleted are printed in red (see Figure A.2). Left-click on a red "tag" opens a new window containing information about the rule that deleted the "tag" and the corresponding report.

It remains to be explained what are the "tags" listed in columns below the positions. Every "tag" is a comma-separated list of one or more values of morphological categories (including lemma) used in *LanGR*. For every position, the list of categories is selected to form a unique key in the set of all displayed tags (for example, if a word form is unambiguous after the morphological analysis, only the lemma is printed).

If the mouse cursor is held over such a "short tag", the full tag is shown in a tooltip box.

[1]Note that the fast *LanGR* is still unable to compile the rules automatically.

Figure A.2: Sample sentence in the web-browser

Appendix B

LanGR functions

The primary objective of this appendix is to present a list of the most useful application level definitions that can be used by linguists to write the rules.

The definitions can be divided into three types: *data type definitions*, *user-defined global variables* and *functions*. Data type definitions contain the internal implementation of what "sentence", "tag", "position" (and others) mean. User-defined global variables are instances of data types, globally defined by a linguist. The most of these global variables are either lists of strings (i.e. instances of the `Vocabulary` data type, see Section 5.3.4), or variables of the `predex` data type that represent "nicknames" of grammatical features, see Section 5.3.12. The functions are defined to use the data types and their instances to express the requirements of linguists.

The rest of this appendix contains the basic list of functions that are directly used by the linguists to express their ideas. The programs in *LanGR* are case-insensitive, hence the identifiers in the following list are equal to those written in lower-case.

In the following table, there is one function header described in every row. The first column specifies the data type returned by the function (if any). The second column contains the header of the function and the third column briefly explains the operation of the function.

Returns	Header	Description
`integer`	`Ambiguity` p	the number of valid (non-deleted) tags on position p
`predex`	p `And` q	logical $p \wedge q$
	`Delete` f `from` p	deletes all tags that match f from position p
`string`	`Emphasize` s	returns the string argument HTML-emphasized
`predex`	p `Implies` q	logical implication $p \rightarrow q$
`predex`	`IsSafe` p	quantifies the formula p by the general quantifier $\forall$

`integer`	`Item` p	applies quantified formula p to the actual position; successful when the position matches (and returns its index), fails if it doesn't match
`integer`	`Facultative Item` p	non-deterministic: either it ignores the argument and returns an `undef` value, or it is equivalent to `Item` p
	`Item SentenceStart`	matches a virtual start of sentence, i.e. a position "before" the first token in the sentence
	`Item SentenceEnd`	matches a virtual end of sentence, i.e. a position "after" the last token in the sentence
`integer`	`Pre-Sentence Item` p	non-deterministic: it is equivalent either to `Item SentenceStart` with `undef` output, or to `Item` p
`integer`	`Post-Sentence Item` p	non-deterministic: it is equivalent either to `Item SentenceEnd` with `undef` output, or to `Item` p
	`Leave` f `in` p	equivalent to `Delete not` f `from` p
`predex`	`MustNotBe` p	equivalent to `not Possible` p
`predex`	`Not` p	logical $\neg p$
`predex`	p `Or` q	logical $p \vee q$
`predex`	`Possible` p	quantifies the formula p with the existential quantifier $\exists$
`predex`	s_1 `Prefix of` s_2	true when the string s_1 is a prefix of the string s_2
`predex`	v `Prefix of` s	true when at least one of the strings from the vocabulary v is a prefix of the string s
	`Report` a_1 a_2 ...	composes a string from its arguments — the string is used by deletion commands (`Delete`, `Unify`) as a deletion message
	`RuleStart`	defines the actions when the rule starts
	`RuleFinish`	defines the actions when the rule ends
`NumberSet`	`Select` p `from` n	uses quantified formula p to select a subset of positions of the set of positions n; only positions that match p are returned

`NumberSet`	`Sequence of` p	uses quantified formula p to match a sequence of positions in the sentence; it non-deterministically decides whether to return the sequence matched so far or whether to try to add the next position to a sequence (see `Item`)
`predex`	s_1 `Suffix of` s_2	analogy of s_1 `Prefix of` s_2
`predex`	v `Suffix of` s	analogy of v `Prefix of` s
`predex`	`Unifiable` n `in` c	tests whether the set of positions n can be unified in the set of categories c; doesn't change the data
`predex`	`Unifiable` p_1 `with` p_2 `in` c	tests whether the two positions p_1, p_2 can be unified in the set of categories c; doesn't change the data
	`Unify` p_1 `with` p_2 `in` c	unifies positions p_1 and p_2 in the categories from the set c
	`Unify` n `in` c	unifies the set of positions n in the set of categories c
	`Unify Conditionally` n `in` c	analogy to `Unify` n `in` c, but performs the unification only if the unification is possible
	`Unify Conditionally` p_1 `with` p_2 `in` c	analogy to `Unify` p_1 `with` p_2 `in` c, but performs the unification only if the unification is possible
	`Unify Unilaterally` p_1 `with` p_2 `in` c	analogy to `Unify` p_1 `with` p_2 `in` c, but modifies only the first argument
`string`	`WordFormOnPosition` p	returns the word form on the position p