A Finite-State Approach to the Inflectional Morphology of the Bulgarian Verb

BA Thesis

Ventsislav Zhechev

ISCL, Seminar für Sprachwissenschaft
Eberhard-Karls Universität Tübingen

Course: *Computational Morphology and Phonology*
Supervisor: *Dr. Dale Gerdemann*

e-mail: ventzislav.zhetchev@sarrio.every1.net
address: Provenceweg 7, Zi. 27
72072 Tübingen, Germany

Contents

Overview

In this thesis I discuss the implementation of a fragment of the Bulgarian inflectional morphology (as described in (Aronson, 1968: 67-91)) using the Xerox Finite-State Toolbox (XFST, described in (Beesley and Karttunen, 2003)). The result of this project is a morphological analyzer/generator for the orthographical or phonological representation of Bulgarian verb forms. Additionally, a transcription system could be derived, matching the orthographical representations of Bulgarian verb forms to the phonetic ones and vice versa.

In chapter 1, I give a short overview of the finite-state morphology theory and present my motivation for choosing the XFST.

In chapter 2, I turn to the linguistic theory that I am implementing, looking into why I chose this theory and how I interpreted it. I also discuss some changes in the Bulgarian language that have taken place since the writing of (Aronson, 1968), which had to be taken into consideration.

In chapter 3, I present the details of the implementation in XFST, first presenting the LEXC lexicon and then the implementation of the morphophonological variation rules in XFST. I give special attention to the motivation behind the design decisions I made in the process of implementation. I also present the testing platform I developed for the project.

At the end of the thesis, I have attached an appendix, which contains the source code of the XFST implementation as well as transliteration tables and notes on the notation I used. There you will also find a few step-by-stem example derivations.

1. Finite-State Morphology

In this chapter, I shortly present the field of Finite-State Morphology and my motivation for implementing a fragment of the Bulgarian inflectional morphology using the XFST.

1.1. Finite-State Technology

Finite-state machines have been known in the field of computation for a long time and their mathematical properties are very well studied. They have been preferred when dealing with string-like data because of the linear speed of computation that can be achieved.

A *finite-state automaton* (FSA) consists of a finite number of *states* and labeled *arcs* (transitions). There are exactly one starting state and zero or more final states. The FSA can be understood as a recognizing device – it either accepts a string or it does not accept it. Given a string to operate on, the FSA is initialized to the starting state. Then, if possible, it traverses the arc labeled with the first symbol in the input string, thus consuming the symbol. After the operation the FSA is in the state to which the arc has led and can further consume symbols from the input string. There may be arcs labeled with ε (epsilon – conventional notation for the empty string) – those arcs are traversed without consuming symbols from the input string. A string is said to be accepted by the FSA if the FSA is in a final state after consuming all and only the symbols from the string. If the FSA is in a non-final state or if there are any symbols left in the input string that cannot be consumed, the string is said to be rejected by the FSA. If the FSA does not have final states, it does not accept any strings and is said to encode the empty language. The example presented here refers to a *deterministic* FSA, i.e. given a current state and an input symbol there is only one arc labeled with this symbol that goes out of the state. An FSA can be *non-deterministic* if there are more arcs with the same label going out of the same state. It has, however, been proved that for every non-deterministic FSA there is an equivalent deterministic FSA.

The theory expands further to include *finite-state transducers* (FST). An FST has the same properties as an FSA, except that the arcs are labeled with pairs of symbols and not with single symbols. While traversing the FST for each symbol from the input string the matching symbol from the arc being traversed is being output. Thus each input string that is accepted by the FST is matched with an output string – a very convenient string-transformation system. All FSTs are inherently bi-directional, i.e. you could consider either the first or the second symbol in each arc label as the input symbol. Further any two FSTs *F1* and *F2* can be *composed* to produce a new FST, which has the input of *F1* as its input and the output of *F2* as its output. In the process of composition only those strings are left in the final FST, for which the output of *F1* matches the input of *F2*. The latter technique allows for the construction of cascades of FSTs, i.e. having as many intermediate FSTs as needed to describe the relation between the input and the output strings.

The FSAs correspond to *regular expressions* and vice versa. That's why they are said to describe *regular languages*. The FSTs, on the other hand, describe *regular relations*, i.e. relations between regular languages.

1.2. Finite-State Morphology

The first computational approach to morphology is the *two-level morphology* from (Koskenniemi, 1983) based on (Kaplan and Kay, 1994). The main idea is that you work with underlying and surface representations and rules that define the relation between these two levels. In the process of compilation the rules are intersected with one another, i.e. they work simultaneously. The rules could be compiled into FSTs, but in order to be able to intersect them the developer should always bare in mind the restrictions on FSTs that are to be intersected – not every pair of FSTs can be intersected. The fact that the rules work simultaneously is also very important: there cannot be any rules that create/destroy contexts for the application of other rules.

In finite-state morphology we also have underlying and surface representations as in two-level morphology. The difference is that in the former approach we have the possibility to add intermediate levels. Thus, instead of having all the rules work in parallel, we have the option to work with a cascade of rules (effectively compiled into FSTs). With such a cascade one could think of the system as working procedurally, applying the rules one after another. This makes the rules simpler and easier to maintain. One could even build modules of rules, which work on a common task, e.g. the alternations occurring in the formation of a single tense form. Besides, there are no restrictions on the form and complexity of the rules as they are compiled into FSTs that are then composed into a single FST. Moreover, as we know, arbitrary FSTs can be composed.

2. The Bulgarian Verb

In this chapter I first shortly go over some general information about the Bulgarian verbal system and then turn to those parts of the theory that are of a particular importance for my project. At the end of the chapter I make a comparison between the theories of (Aronson, 1968) and (Scatton, 1984) from a practical point of view.

2.1. General Details

In Bulgarian there is a great number of verb forms which can be produced from a single stem by inflectional processes. In addition to the overt person and number distinction, the Bulgarian verbs are also marked morphologically for tense, aspect and mood. The participles, derived from the verbs, are additionally marked for gender in the singular – following the same patterns as the adjectives. The verbal substantive is always in neutral gender; there are imperative verb forms for 2nd person singular and plural and there is a gerund verb form, which is not marked for person and number.

Some scholars argue that there are nine 'tense' forms in Bulgarian, and others say that there are even more. The total number of possible combinations of tense, aspect and mood (all three categories being morphologically marked) is 24, although most of those forms are complex, i.e. they consist of one or more auxiliaries plus a participle derived from the verb stem. For my project I disregard the complex forms and will consider only the simple ones – consisting of an inflected verbal stem only.

The verbal forms I am analyzing are: Aorist past tense, Imperfect past tense, Non-past tense (corresponding to present tense, but also used in the formation of complex tense forms, e.g. future tenses) and the corresponding participles; the passive participle; the gerund form; the imperative form and the verbal substantive.

For my project I follow (Aronson, 1968)'s theory. According to him the Bulgarian verb is composed of a (prefixed) stem and a desinence. The desinence may be terminal or non-terminal. A complex desinence consists of one or more non-terminal desinences and a terminal desinence. (Aronson,

1968) takes the base form of the stem either from the third person plural non-past without the personal desinence or from the aorist participle without the terminal desinence and the non-terminal desinence *-l-*.

However, in some respects his data are outdated (i.e. incorrectly placed stress, /e/ occurring where /ja/ should have been). In such cases I change the rules and/or the lexicon data so that they correspond to the contemporary Bulgarian language.

2.2. Types of Verbal Stems

The Bulgarian verbal stems can be divided into two groups according to (Aronson, 1968), depending on the final vowel of the base form:

2.2.1. Vocalic stems

These are stems ending in a vowel. This group is itself divided into two subgroups: non-truncating and truncating vocalic stems.

The non-truncating stems end in /a/ and the stress falls on a non-final syllable. Members of this group are derived stems with suffixes such as *-íra-*, *-isa-*, *-uva-*, etc. (*telefon-íra* – telephone, *izograf-ísa* – paint icons, *interes-úva* - interest); derived imperfectives with the suffixes *-áva-*, *-va-*, *-a-* (*o-xladj-áva* – cool/chill, *na-pís-va* – finish writing, *plášt-a* – pay); and non-derived verbs (*gléda* – look, *do-čáka* – wait).

The truncating vocalic stems end in /é, í, i, á, a/. The stress may fall on the final vowel (β-stress) or on a non-final vowel (α-stress). Members of this group are all verbs in *-na-* and some non-derived verbs in *-a-* that should be listed. If the stem of a verb from this class has no internal root vowel, it is called a zero-stem. Examples of verbs from this group are *svárji* – find, *písa* – write, *b#rá* – take, etc.

2.2.2. Consonantal Stems

The base forms of the stems in this group end in a consonant. Here we can also subdivide the stems into truncating and non-truncating.

The base forms of the truncating consonantal stems end in /j/ or /m/. Examples are *pjéj* – sing, *vzém* – take, etc.

The stems which end in consonants other than /j, m/ are non-truncating. Examples are *bod′* – gore, *pek′* – bake, etc.

2.3. Rules of Conjugation

In this section I discuss the morphophonological rules involved in the generation of the verb forms in Bulgarian following (Aronson, 1968). The main stress is on the relevance and thematic grouping of the rules, but I also point out where the theory deviates from the contemporary Bulgarian.

First I look at some "non-specific" rules that should be considered and then I go through the "specific" rules, divided in several groups. 'Non-/specific' refers to whether the application of the alternation rules is restricted to the verb class or may occur in any word class.

2.3.1. Non-specific Alternation Rules

Preliminary to the rules of conjugation one should take into consideration a few other facts in order to have a complete system. The first to be considered are the phonological restrictions on the occurrence of consonantal clusters in Bulgarian (Aronson, 1968: 36-40). An example is the following alternation:

(1) rast-na → rasna

Here the *t* is deleted as the *stn* cluster is phonologically impossible in Bulgarian.

Very important are also the so-called *automatic alternations* (Aronson, 1968: 44-46) – alternations that are conditioned only phonologically. (Aronson, 1968) uses this term to refer to the rules of assimilation and dissimilation, a restriction on the occurrence of stem final /r, l, v, m/ and a rule defining the realization of morphophonemic sequence <C'e>. The first two rules, defining the voicing/devoicing of consonants, are only important for deriving the correct phonetic representation, as those alternations are not represented in the Bulgarian orthography.

The third rule has a rather vague definition and if implemented directly, produces wrong results. But in the fragment of the inflectional morphology that I am covering, there is only one context where this rule applies: the formation of the masculine past participles. This makes the use of a specialized version of the rule possible.

Overall importance for the correct derivation has the last rule of automatic alternation. (Aronson, 1968) gives two possible realizations of the morphophonemic sequence <C'e> – /Ce/ and /C'á/ – and describes the contexts in which each of them occurs. As both realizations appear in the contexts triggering other rules, this automatic alternation affects the execution flow of the whole system.

After these preliminaries I turn to the rules of conjugation. The presentation is divided into five parts, following the grouping in (Aronson, 1968): general rules; desinences; non-past forms; aorist forms; anomalous formations.

2.3.2. General Conjugation Rules

There are four general rules described in (Aronson, 1968: 68-69), which apply in the process of conjugation. For the stem typology refer to section 2.2.

The first rule – the *stem truncation* rule – defines the formation of the two types of stem forms that are used further for the derivation of the non-past and aorist series of forms. This stem forms are usually referred to as the *non-past* (*imperfect*) and the *aorist* stems and are described in sections 2.3.4 and 2.3.5. For the formation of the non-past stems the final vowel of the stem is deleted in truncating vocalic stems before all non-truncating vocalic desinences. For the formation of the aorist stems, on the other hand, the final consonant in truncating consonantal stems is deleted before all truncating vocalic desinences. The difference between truncating and non-truncating vocalic desinences is explained in the next paragraph.

The second general rule of conjugation is also a truncation rule – the *desinence truncation* rule: "all desinences with initial vowel drop that vowel after non-truncating vocalic stems". Vocalic desinences, thus, are the desinences with initial vowel. Some of them lose this vowel after truncating vocalic stems as well. These are the truncating vocalic desinences.

The next rule defines the consonantal alternation /k, g, x/ → /č, ž, š/. This alternation occurs on the boundaries between stem and desinence or between desinences – where the sequence of a velar plus an acute vowel (/e, i/) is not permitted.

The last rule in this group accounts for the general stress distribution. By default the stress falls on the same syllable as in the base form. In case the stressed vowel is truncated, or the stem has β-stress, the stress falls on the immediately following desinence. There are, however, some desinences that cannot be stressed – the atonic desinences. In this case the stress has to move back and falls on the last syllable of the truncated stem.

2.3.3. Selection of Desinences

The tables listing the possible desinences can be found in (Aronson, 1968: 69-70).

The desinences are divided in two groups: non-terminal and terminal desinences. The non-terminal ones include the tense and participial form markers and the verbal substantive marker. The terminal desinences are the person markers of the past and non-past, the person and number markers for the participial forms (which are the same as for the adjectives) and terminal desinences for the gerund and the imperative forms.

The rules used for the formation of the forms from the next two groups are very similar to the general rules I discussed above: there are rules describing consonant assimilation processes; there are rules describing stress shifts; and there are rules describing the selection of desinence variants. I am not going into the details of these rules, as this is not the goal of my project. Brief comments on the most interesting rules are given alongside their implementation in section 3.1.2.

2.3.4. The Non-past Series of Forms

The forms in this group are derived from the *non-past* stem – which is either a truncated vocalic stem or a non-truncated (full) consonantal stem. These forms are the *non-past tense*, *imperfect tense* and *imperfect participle*, *imperative*, *active participle* and *gerund*.

2.3.5. The Aorist Series of Forms

For the derivation of the forms in this group, the *aorist* stem is used – either a non-truncated vocalic stem or a truncated consonantal stem. The *aorist tense*, *aorist participle*, *passive participle* and *verbal substantive* can be formed from this stem.

2.3.6. Anomalous Formations

(Aronson, 1968) lists seven groups of anomalous formations, which can be found on pp. 73-74. I will only say that these anomalous formations cover the derivation of the forms of the auxiliary verbs *štjá-* "want" and *sým/býda* "be". Their detailed discussion follows in section 3.1.3.

I have found a small deviation from the contemporary Bulgarian here: (Aronson, 1968) claims that the 1st and 2nd person plural imperfect past forms of the verb *štjá-* are irregular – *štéxme* and *štéxte* instead of the regular *štjáxme* and *štjáxte*. This irregularity, however, has disappeared from the standard contemporary Bulgarian and remains only in some dialects.

2.3.7. Example Derivations

In (Aronson, 1968: 74-84) you can find lists with example derivations for stems from all major gourps. Unfortunately, not all possible forms are listed for each stem. That is why in Appendix D I have included lists of all possible derivations for two stems – in order to show the variety of forms that can be derived from a single stem.

2.4. Other Theories

In this section I am actually looking at only one other theory – the theory of (Scatton, 1984). There are other theories on Bulgarian inflectional morphology, but they are descriptive in nature and thus not suitable for computational implementations.

(Scatton, 1984)'s theory differs in many ways from the one of (Aronson, 1968). The main difference is in the goals of the two theories – while (Aronson, 1968) concentrates on the inflectional morphology, (Scatton, 1984) is trying to develop a complete theory covering inflectional as well as derivational morphology. Because of this, the latter tries to minimize the number of rules while maximizing their reusability. This, unfortunately, makes it more difficult to understand the ordering of the rules and their linguistic meaning. It also leads to the creation of numerous exceptions to each rule. In (Aronson, 1968), on the other hand, most of the rules are crafted especially for a specific formation. Although this leads to a slightly greater number of rules, the rules have fewer exceptions and are easier to implement – one reason for choosing (Aronson, 1968)'s theory for my project.

Another crucial difference between the two theories is the choice of base forms. While in (Aronson, 1968) an explicit algorithm is given for the derivation of the stems' base forms, (Scatton, 1984) does not say anything on the issue. Because of this, a lexicon built according to (Aronson, 1968)'s theory will be easier to populate and maintain – both manually and automatically.

In this chapter, after giving some general information about the properties and singularities of the Bulgarian verbal system, I described the details of the linguistic theory I am implementing in this project.

At the end of the chapter I made short comments on my motivation for choosing (Aronson, 1968)'s theory – namely that could easily be implemented in a computational framework.

3. LEXC/XFST Implementation

In this chapter, I look into the technical details of the LEXC/XFST implementation and talk about some of the difficulties I encountered. I also discuss the issues of testing the implementation and expanding the lexicon.

For the transliteration rules I am using check Appendix A; the lexicon is in Appendix B and the rules implementation in XFST is in Appendix C. In Appendix D I have included a few example derivations, described step-by-step.

For the general understanding of the system, it is very important to bear in mind that it uses Cyrillic letters internally. That is to say, each phoneme (or just sound – depending on the context) is represented by a Cyrillic letter. I also use the letter j, which does not exist in the Bulgarian alphabet, to mark either palatalization or the glide /j/. Of course, I could have chosen to use Latin letters or even IPA characters, but the use of Cyrillic letters makes it easier to extend the lexicon using existing resources in Cyrillic.

It should be mentioned that I am using the non-commercial version 8.4.3 of the XFST toolbox. This version is a requirement, as earlier versions do not have full support of the Unicode character encoding – the one used in my implementation.

3.1. System Design

In this section, I first describe the LEXC lexicon in detail and then turn to the implementation of the morphophonemic alternation rules in XFST, giving special attention to the more complicated cases. Finally, I present the overall architecture of the system, detailing the interconnections between its different elements – the lexicon, the rules and a few auxiliary FSTs.

3.1.1. Lexicon

The lexicon describes the underlying or analysis forms of the verbs. On the upper side, it has the base form with the analysis features attached; and, on the lower side, it in addition has all the internal features needed for the correct generation. For convenience, I will call the upper side strings in the lexicon the input strings and the string on the lower side – the output strings. Following is the structure of the LEXC lexicon.

The first part of the lexicon is the *Definitions* part. There I define variables corresponding to different flag diacritics (henceforth *flags*). This is done in order to spare the lexicon writer the need to remember the complex syntax of the flags in LEXC – the variable names are more concise. The first group of variables defines the flags that require the existence of a single prefix. The names of these variables correspond directly to the Latin transcription of the prefixes. The second and third groups define the flags used when a double prefix occurs. The variable names are the same as in the first group with an *f* added to mark the first element of the double prefix and an *s* – to mark the second element. From these two groups one can construct an arbitrary double prefix. For the rare cases when a triple prefix occurs, its second and third part are treated as a single prefix and added to the third group of definitions. The last two small groups of definitions refer to the flags used for *aspect* and *transitivity* marking. I say more on the use of these defined variables further in this section, when discussing the other parts of the lexicon.

The rest of the lexicon is divided into *sublexicons* (also called *continuation classes*). The entries in the sublexicons have the following general form:

```
entry         continuation_class;
```

An entry can either consist of a single string, which is compiled in an FSA or an identity FST; or of an *upper:lower* pair, which is directly compiled as the upper and lower side of an FST; or of a regular expression in angle brackets (<…>), which is being compiled using the XFST regular expression compiler.

In the process of compilation, the sublexicons are considered starting with the one with the special name 'Root'. For each entry in a sublexicon, its contents is concatenated with the string[1] that has accumulated that far (in

[1] We can talk of strings when the lexicon is compiled into an FSA. When, on the other hand, the lexicon is compiled in an FST, we should be talking of upper:lower string pairs. In this case, the upper string in the entry considered is concatenated to the upper string so far and the lower string in the current entry is concatenated to the lower string so far.

the case of the 'Root' lexicon it is the empty string) and its corresponding continuation class is being followed to the sublexicon it represents. Then the procedure continues recursively for all entries of that sublexicon. The special '#' continuation class represents the end of the string and terminates the current branch of the recursion. In this way, an FSA or FST is built, representing all possible strings that could be produced from the lexicon data. More information on the topic can be found in (Beesley and Karttunen, 2003).

In my case the lexicon will be compiled into a transducer, so most of the entries in the sublexicons are regular expressions defining FSTs. The first sublexicon in the lexicon file is the `Root` lexicon. From it the system can either start of with adding an `«Pref»` tag on the lower side of the transducer and continue to the `Prefixes` sublexicon, or alternatively bypass the prefixes altogether and continue in the `StemTypes` sublexicon.

Next comes the `Prefixes` sublexicon. Here all the 20[2] possible prefixes are listed in the form of entries, which add the prefix to the string and set the value of the `Prefix` flag to the corresponding prefix. The entries from this sublexicon can represent either a single prefix or the first part of a double (or triple) prefix. Each entry then leads to the `SecPrefixes` sublexicon. Here all possible prefixes are listed once again, with the difference that this time the `SecPrefix` flag is being set. These entries represent the second part of double prefixes. Additionally, there are entries for treating cases when a triple prefix occurs – they are effectively double prefixes, representing the second and third part of the triple prefix. These entries should be entered manually when necessary during the extension of the lexicon. There is one more entry in this sublexicon, which is empty and thus allows for the generation of single prefixes. All entries in this sublexicon lead to the `Stems`

[2] The standard analysis defines 19 verbal prefixes for Bulgarian, two of which – *v-* and *s-* – alternate with *vy-* and *sy-* respectively in certain contexts. The problem with such analysis is that whereas the alternation *v-* → *vy-* can be easily defined and there are no exceptions, the alternation *s-* → *sy-* may or may not occur in exactly the same contexts (e.g. *s-gromoljás-am* but *sy-gradj-á*). Therefore I have included *sy-* as a prefix on its own – which brings the number to 20. The list of prefixes was taken from (Scatton, 1984).

sublexicon, which has a single entry adding a ÷ on the lower side – a boundary marker between the prefix and the stem. The continuation class for this entry is `StemTypes`.

The `Prefix` and `SecPrefix` flags, being set as they are in these sublexicons, make it rather straightforward to define the lists of possible prefixes for each verb – each such list will be a disjunction of flags which require the existence of one or another prefix flag – effectively requiring the existence of one or another prefix in the string accumulated so far. Had the flag mechanism not been available, one would have had to add each prefixed stem as a separate entry – copying the verbal stem each time.

Obviously I am using the flags for something they were not designed for (cf. (Beesley and Karttunen, 2003) Chapter 8) – there is no "separated dependency" between the prefix and the stem. This case could be described as "backward dependency", as the stem selects the prefix that should precede it. With the use of flags, though, the lexicon will be smaller and the addition of new stems will be easier – a sufficient motivation for applying it.

No matter what path the system takes through the sublexicons discussed so far, it will eventually come to the next sublexicon in the file – the `StemTypes` sublexicon. There are six entries in this sublexicon that define the values of the `asp` (aspect) and `trans` (transitivity) flags and lead to the respective sublexicons where the stems with the described properties are listed: the first entry defines the *non-transitive perfective* stems; the second – the *non-transitive imperfective* stems; the third – the *transitive perfective* stems; the fourth – the *transitive imperfective* stems; the fifth – the *non-transitive derived* stems; and the sixth – the *transitive derived* stems. The last two are not marked for aspect at this stage. It is important to explicitly state these properties of the stems, as they cannot be deduced from the form of the stem[3].

[3] Actually the aspect could almost always be deduced for the derived perfectives and imperfectives, but the primary perfectives and imperfectives have to be explicitly marked.

All of the stems-listing sublexicons have the same structure, so I am not referring to any one of them in particular. Each entry is defined as a regular expression. First, it defines the verbal stem, e.g.:

```
<{дерз´аj} …
```

Some basic stem variation is marked directly at the stem definition:

```
<{({благо}){ух´аj} … !smell sweet
!the stem is either ух´аj or благоух´аj
```

The next thing is to define the stem according to (Aronson, 1968)'s classification. This is done by adding a combination of the internal features `«Trunc»`, `«VS»` and `«CS»` to the output string:

```
… 0:[«Trunc» «CS»] …
!a description of a truncating consonantal stem
```

This part is not present in the derived stems, as it is introduced later together with the desinence.

The last part in the stem description is the list of possible prefixes. It is stated as a union of defined variables, each of which represents a flag requiring the presence of the corresponding prefix. Thus, each entry produces as many paths (or, in other terms, basic forms) through the lexicon as there are prefixes available for it. If the stem can be non-prefixed, the variable `dp` is added to the union to add a path through which only a non-prefixed stem can go. When the stem cannot combine with any prefix, this is the only element in the prefix list. In Bulgarian, prefixation is a derivational process, which changes the aspect from imperfective to perfective (for primary imperfective stems). Because of this, the prefixes are usually grouped together and for all of them the `naimp` flag variable is added, effectively changing the aspect to perfective for the prefixed forms. The prefixed forms also often differ in terms of transitivity from the non-prefixed forms. These differences should also be reflected in the stem description and this is done in the following way: when the non-prefixed stem is transitive, the prefixes for which a non-transitive form is derived are grouped together and the `nto` flag variable is added (turning transitivity off); when the non-prefixed stem is non-transitive, the prefixes for which a transitive form is derived are grouped together and the `pto` flag variable is added (turning transitivity on). Here are a few examples:

```
… [[[iz|raz|s]naimp]|dp]> …
… [[[[[za|po]nto]|iz|fpo siz|pre|raz]naimp]|dp]> …
… [[[za|fpo sza]pto]|po|dp]> …
… dp> …
```

For the non-suffixed stems, the continuation class `Des` is used - pointing to the sublexicon, from where the desinence definitions start. For the suffixed stems, there are eight sublexicons defined and each stem points to the corresponding sublexicon. Each of these sublexicons has a single entry. These entries first add a morpheme boundary between the stem and the suffix and then describe the corresponding suffix and the common properties of the stems that have this suffix, e.g.

```
<0:÷ a 0:«VS» paimp> …
!the suffix a - non-truncating vocalic stems - always imperfective
```

There is a common continuation class for all suffixes – `Des`. This is also the sublexicon to which you come directly from the non-suffixed stems. The only entry it has adds the stem-desinence boundary to the output string and continues to the `DesTypes` sublexicon.

The `DesTypes` sublexicon, in turn, has seven entries. Two of them give all information for the derived forms and have the continuation class `#` – the end of the string. These entries describe the desinences for the gerund form and for the verbal substantive. As all other entries in the sublexicon, they first add the corresponding analysis features both to the input and output string (here `«Ger»` and `«VSubst»` respectively). Then they add the desinence to the output string and end with a requirement to the stems from which the forms can be derived – the stems should be imperfective (marked by the `raimp` flag variable). There are two other interesting entries. The first is the imperative entry, which in addition to the analysis feature `«Imper»` adds the non-terminal desinence to the output string and then points directly to the terminal desinence. The other one is the entry for the passive participle, which also adds the non-terminal desinence to the output string and in addition has a requirement to the stems, which can derive this form – they should be transitive (marked by the `rto` flag variable). It points to the sublexicon with the participial terminal desinences. The rest of the entries have continuation classes, pointing to the corresponding non-terminal desinences.

There are three sublexicons, defining non-terminal desinences – for non-past, imperfect and aorist forms. Each has two entries – one for the normal and one for the participial forms, pointing to the corresponding sublexicon with the terminal desinences.

At the end of the lexicon, there are five sublexicons, defining the terminal desinences: for the *non-past*, *imperfect*, *aorist* tenses; for the *participial* forms and for the *imperative* forms. As these are the terminal desinences, all entries have `#` as their continuation class.

In this chapter, I described in detail the internal structure of the LEXC lexicon. The important points, which should be considered when extending the lexicon, are the following:

1. The base sublexicon for the new stem should be chosen, i.e. one should know whether the stem is *perfective/imperfective*, *transitive/non-transitive*, *derived/non-derived*.
2. Then one should classify the stem according to (Aronson, 1968).
3. Further, one should compile the list of possible prefixes, considering which, if any, change the *aspect* and/or *transitivity* of the stem and mark them respectively.
4. Finally, one should choose the continuation class – either `Des` for the *non-suffixed* stems, or the continuation class for the corresponding suffix.

In my opinion, it is possible to do these tasks (semi-)automatically, using existing verb listings or dictionaries for Bulgarian, as I have already defined all irregular stems from (Aronson, 1968) .

Finally, I add some statistics for the lexicon: there are 341 stem entries, amounting in 1773 base forms, when combined with the prefixes; the total number of verb forms produced by the system is 56253, having 46286 distinct orthographical and 47529 distinct phonetic representations.

All stems defined in the lexicon are taken from (Aronson, 1968). The prefix lists were extracted manually from the offline version of (Angelov, 2005). The well-formedness of some of the forms was checked in (Pashov and Parvev, 1979).

3.1.2. Rules

In this section I describe the XFST implementation of the morphophonological rewrite rules from (Aronson, 1968). In addition I present the motivation behind the design decision for each rule. You can find the rules I refer to in Appendix C – in file `bop-lrules.xfst`.

The rules refer to different predefined variables. Most of these variables are defined in the same file and a few are defined in `bop-urules.xfst`. The general naming convention for the rule variables is that they start with a combination of digits and letters, corresponding to the section in which they are defined in (Aronson, 1968: 68-73). The rest of the name is a short description of the function of the rule. There are three rules, which are described earlier in the book, whose names start with a zero. The general format of the rules is

```
source -> replacement || left_context _ right_context;
```

It is possible to have multiple `source -> replacement` parts, separated by commas; and it is possible to have multiple context parts, separated by commas. To mark the string boundaries in the contexts the special symbol `.#.` is used. In XFST it is also possible to have multiple replace rules, working in parallel – marked by separating them with double commas (`,,`). I will describe any differences in the notation on the go.

I first look at the 'zero' rules. The first one – `0PrefixExtention` – is a phonological rule, describing the prefix alternation *v-* ⇝ *vy-*:

```
define 0PrefixExtention [..] -> ъ || .#. «Pref» в _ ÷ [в|ф|[(´) о]| ↩
                                                    [п C [V - AcuteV]]];
```

This is a straightforward rule, inserting a ъ every time a matching context is found. An insertion is implemented as the replacement of an epsilon (`[..]`) by something (in this case ъ). The left context of the replacement contains a prefix feature (`«Pref»`) followed by the prefix *v-* (в) at the beginning of the string (marked by the .#. symbol). The right context contains a morpheme boundary (÷) followed by either one of the elements of the disjunction. The last one, for example, means that there is a п phoneme, followed by an arbitrary consonant (the variable C) and an arbitrary non-acute vowel (all strings from the variable V minus the strings in the variable AcuteV). Most of the other rules work in exactly the same way, so I am not describing them in detail.

The other 'zero' rule – 0CjEAlt – implements the automatic alternation describing the distribution of the two possible realizations of the morphophonemic sequence <C'e>. You can find more information in section 2.3.1 and in (Aronson, 1968: 45). There is nothing special to this rule too – it translates directly Aronson's description into XFST.

Next I look at the general conjugation rules, applying to all verb forms. The first of them – 1AStemTrunc – is an example of doing two replacements in parallel: it does the truncation both for the vocalic and for the consonantal stems in the corresponding contexts. The truncation (deletion) is implemented as something being replaced by an epsilon ([]). The next rule – 1BDesTrunc – describes the general desinence truncation rule. Here you may notice the internal feature «Exc5». The use of the 'exception' internal features is described in detail in section 3.1.3 and short descriptions can be found in Appendix A. Another simple rule is the one, describing the velar alternations */k, g, x/* ⇝ */č, ž, š/* – 1CVelAlt. More interesting is the implementation of the general stress distribution rule from (Aronson, 1968: 69). According to the rule, the stress could stay in place, move forward or move backward in certain contexts. One way of implementing this rule is to develop a series of replacement rules, which delete the stress from a wrong position and then insert it in the right one. In this case, though, it is very difficult to find the correct contexts for such a series of rules. That is why I chose another option – to implement the rule directly as a transducer, which deletes the stress in one place and inserts it into another. The actual implementation is a disjunction of three transducers: the first one describes a forward stress shift and the context in which it occurs; the second one describes a backward stress shift and the context in which it occurs; the third disjunct describes the stress assignment in the case of zero stems with stressed zero vowel (this rule is not mentioned directly in (Aronson, 1968)); the last disjunct describes the backup case – when no stress shift occurs. The FSTs being thus defined, no string can pass through more than one of them at the same time, i.e. every input string for this rule gets a unique output matched. In addition, no input strings are blocked by the rule.

The next group of stems operates exclusively on the non-past stems and defines the formation of the non-past series of verb forms (*non-past tense, imperfect past tense, imperfect participle, imperative, active participle, gerund*). I only look at the most interesting and complex cases here – starting with the first two replace rules. They together implement one stem alternation rule from (Aronson, 1968). The implementation had to be divided into two parts, as the second of the two replace rules could be instantiated in contexts in which a replacement from the first replace rule should have been executed (for example */g/* ⇝ */ž/* instead of */zg/* ⇝ */ždj/*).

An apparently interesting decision is the implementation of the terminal desinence selection rule in (Aronson, 1968: 70-71). There are several possible ways of implementing it. The first possibility is to add only a verb form marking feature tag in the lexicon and then using rules to insert the correct desinence in each case. A second possibility is to let the lexicon overgenerate, adding any possible desinence, and then to implement filters that rule out the incorrect cases. The third possibility – the one I chose – is to choose one desinence variant as the base form and to replace it with other variants when needed. In my opinion, this variant was the simplest to implement and makes the system more straightforward.

Another rule worth mentioning from this group is the stress shift rule for the formation of the imperative. It is implemented – as the previous stress shift rule, discussed above – as a disjunction. Here, there are only two disjuncts: the first describes the stress shift and the second lets through all strings in which no stress shift occurs (in this case – non-imperative verb forms).

The last group of replace rules implementing (Aronson, 1968)'s theory defines the aorist group of verb forms (*aorist past tense, aorist participle, passive participle, verbal substantive*), operating exclusively on the aorist stems. From this group I also discuss only the most interesting rules. The first stress shift rule from this group – `4A1StressAlt` – differs a little from the other stress shift rules I discussed so far – instead of a normal union it uses priority union. The only reason for doing it this way is that it results in a smaller FST for this particular rule, without changing its functionality. The other stress shift rule – `4D1StressAlt` – works as 'usual'.

There are two other rules I discuss – `4C2DesSel` and `4D1StemTrunc`. They both had to be implemented as two-part rules for similar reasons. According to (Aronson, 1968), there are some obligatory changes occurring in these cases and some optional. Of course, this could not be implemented with a single rule – that is why I had to write a rule for the obligatory changes and a rule for the optional changes in each case.

In addition, you will find a rule in this group, which is completely 'exceptional', i.e. it only obtains in exceptional cases, containing the «Exc3» internal feature.

This concludes the description of the alternation rules' implementation. As you have seen, most of (Aronson, 1968)'s rules can straightforwardly be translated into XFST's replace rules notation. Some of the rules, though, are more difficult to implement – requiring crafting of precise regular expressions.

3.1.3. Difficulties

Here I discuss the main difficulties I encountered during the implementation of (Aronson, 1968)'s rules. I also discuss the exception handling techniques I used.

I start with some notes on the flag diacritics. As I have described in section 3.1.1, I am using flags in the lexicon, in order to simplify its structure. The problem is that the flags interfere with the application of the replace rules and the composition of FSTs[4]. Therefore, a way should be found to 'ignore' them in such cases. My solution was to use the XFST command `eliminate flag` to clean the lexicon FST from the flags after it is compiled. The resulting FST encodes the same input and output languages, but does not have flags in it, thus allowing for the problem-free application of the replacement rules. The other solution, described in (Beesley and Karttunen, 2003), was to instruct the system to treat the flags as epsilons (empty strings) so that they do not interfere with the composition and replace rules. In my case, this move produced a bigger and slower FST – that is why I chose the former solution.

The next problem I encountered was with the definitions of the contexts in which the rules applied. In (Aronson, 1968: 32-34), we find tables in which he defines the phonological inventory of the Bulgarian language. The phonemes are classified according to different features (e.g. *grave* vs. *acute* vowels and consonants; *compact* vs. *diffuse* consonants etc.). Unfortunately, he does not refer to these features when defining the contexts of the morphophonological rules. There he refers to more standard[5] features, e.g. *velar*, *palatal*, *labial* etc. – features, not present in his classification. This made the implementation of the rules more difficult, as I had to check other sources (e.g. (Scatton, 1984)) and eventually adjust the data to match (Aronson, 1968)'s classification and my knowledge of the language.

A further problem arose with the ordering of the rules. There is no explicit ordering stated in (Aronson, 1968). In some footnotes he states that one rule is applied before another, but this is not enough to deduce the complete ordering. Thus, the only way to find the rule ordering was to experiment with it and, in some cases, even tweak the contexts of some of the rules, so that all transformations are instantiated correctly.

The last significant problem was the exception handling. I considered several ways of implementing this. One possibility is to prepare an explicit list of FSTs, defining the input and the output strings for the exceptional

[4] cf. (Beesley and Karttunen, 2003: 361-362)

[5] i.e. widely used nowadays

cases, and then combine it with the general FST using priority union. In my case, however, this is an impractical solution – there are a great number of exceptional cases, which can be classified into groups with similar characteristics, e.g. the same alternation affects several different stems or several different verb forms of the same stem. That is why I chose another option: I implemented replacement rules, which replaced the erroneous outputs from the lexicon with the correct exceptional forms. Then I composed these transducers in-between the lexicon and the 'normal' replace rules. This has the advantage, that I know what verb forms exactly I am working with (looking at the internal features included) and thus can implement the rules so that they operate on whole exception classes:

```
[[«Trunc» -> [] || ´ ?* [{зг}|{ск}] a _ «VS» $NPS]| ↩
            [[..] -> «Exc2» || ´ ?* [{зг}|{ск}] a «Trunc» «VS» _ $NPS]]
![zg|sk]«Trunc» -> [«NTrunc»|«Exc2»]
```

This also makes it easy to delete some verb forms for certain stems, thus effectively preventing their production:

```
[{c´a} -> [?-?] || .#. _ M+ ÷ ÷ [~$«NPast»|$«Part»] .#.]
!sá- has only NPast forms
```

In some cases, the exception is that some rules do not apply when they should (or apply when they shouldn't). Instead of listing those cases, I added some 'exceptional' features to the list of internal features. The presence (or absence) of those features controls the execution of some rules – to model the exceptions. Altogether there are five such features – each serving a different purpose. Some of these features are added to the output string using explicit exception handling rules:

```
[{държ´a} -> [{дръж´} «Exc1»] || _ M+ ÷ ÷ $«Imper»]
!dyrží -> drýž
```

and some are added to the output string in the lexicon – together with the stem definition:

```
<{гнетj´e} 0:[«Trunc» «VS»] [[[na|u]0:«Exc3» pto naimp]|dp]> Des;
!gnetjé- -> oppress
```

I discuss the specific use of each exceptional feature below:

The «Exc1» feature is being introduced only through exception handling rules. It is utilized to enforce the application of alternation rule

3C2aSgDesAlt – describing the singular desinence distribution for the formation of the imperative forms.

The «Exc2» feature is being introduced through exception handling rules, as well as directly in the lexicon. It is utilized to prevent the application of rule 3A2aNTDesFilter – describing the distribution of the non-terminal desinences of the non-past tense.

The «Exc3» feature is being introduced only in the lexicon and only for prefixed forms. It allows the application of the exception-only rule 4D1StemAlt – describing an exceptional depalatalizing in the process of formation of the verbal substantive.

The «Exc4» feature is also being introduced only in the lexicon. It is used to add an additional context for the application of the stress shift rule 4A1StressAlt – in order to describe the exceptional stress shift in some cases.

The «Exc5» feature is only used for the stem variant c´a- of the verb *be*. It prevents the application of the general desinence truncation rule 1BDesTrunc for this stem.

3.1.4. Putting it All Together

In this section, I present the interoperation of the different modules of the system. I will first discuss what auxiliary FSTs are used and how they are built. Then I will show how they combine with the lexicon and the replacement rules to produce the orthographical/phonetical analyzer/generator. Finally, I will show how some other modules can be constructed, using the FSTs I already have.

One of the additional modules that are needed for the construction of the system is the transliteration module. Actually, I could have left it out, but one of my goals was to have the stems transliterated in the analysis strings – making them look like (Aronson, 1968)'s stems and making the analyses more accessible to the general public. This module is implemented as a 'big' parallel replacement that changes the Cyrillic letters into Latin letters. To enable the transliteration module to operate bi-directionally, on the top of the replacement rule a filter is composed that throws away all non-Cyrillic characters. In addition, to make the stems look tidier, another replacement rule is composed on the lower side of the transliteration module to create stressed vowels out of separated stress and vowels (e.g. ´a is replaced with á). This rule can be found in Appendix C in file bop-urules.xfst.

Very similar to the transliteration module is the IPA transcription module, with the only difference that it replaces the Cyrillic letters with IPA phonetic symbols. I have not included the implementations of the transliteration and IPA modules in Appendix C. For the correspondence between the Cyrillic letters and the Latin letters of the IPA symbols, refer to Appendix A.

The rest of the auxiliary FSTs are defined at the end of the file bop-lrules.xfst. There you fill find a group of replace rules, combinations of which are used to create the Ortho and Phono FSTs – the former used for the generation of the orthographical representation of the verb forms, and the latter for the generation of the phonetical representation. These replace rules operate on the string produced from the phonological alternation rules, e.g.

```
«Pref»от÷гад´а«Trunc»«CS»÷÷«Aor»¬х÷«3P»«Pl»°а
guess-«Aor»«3P»«Pl»
```

The first one – remstress – is used to remove the stress marking from the resulting string – stress is not marked in the Bulgarian orthography. The next – repstress – repairs the incorrect placement of the stress in a few verb forms (derived from the stem -´jd-). stress is used to make sure that the stress comes before[6] Cyrillic vowels like я or ю in case that you choose to leave the stress marking in the orthography. Further, you see a completely orthographical rule – finalschwa – saying that no ъ can appear word finally in the orthography. The next four rules are used to delete the markup – the morpheme boundaries, the zero vowel markings, the markup for truncating desinences and the internal and analysis features – in that order. Then there are two more orthographical rules, treating the representation of sequences

[6] as opposed to *in the middle of* – the reason is that this vowels are not used inside the system, but are instead represented as a palatalization mark and a or y respectively – what they actually are

of a palatalization marker and a vowel or a consonant in the orthography. The last rule in this group is a completely phonetic rule – describing the process of assimilation of voicing in consonants – a very common process in Bulgarian.

Using the FSTs described in this thesis so far, several different systems can be produced. I will start with describing the two that are the main goal of this project: a generator/analyzer for orthographical representations of Bulgarian verbs and a generator/analyzer for phonetic representations of Bulgarian verbs. In the cores of both systems stays the `Rules` FST that you can find in Appendix C in file `b`. To derive this FST the alternation rules have to be composed with each other in the correct order. These rules operate on the output of the lexicon, so it should be composed on top of them. Between the lexicon and the alternation rules, though, the exception handling rules are composed – as described earlier in section 3.1.3. So that the `Rules` FST is able to produce the correct strings on the upper side, the transliteration module should be added to the upper side. Its inverted version used in the composition (as well as the inverted version of the 'tidy up' rule), as it is implemented to transliterate Cyrillic into Latin and here we actually have the Latin as the input.

After we have the `Rules` FST defined, we can compose either the `Ortho` or the `Phono` FST on its lower side, in order to produce the orthographical or phonetic representation respectively.

Another system that can be defined is a 'transcription' module relating the orthographical representations to the phonetic ones and vice versa. At first, one could think that this can be done using only the `Ortho` and `Phono` FSTs. At a second glance, though, one will notice that stress is not marked in the orthography, i.e. there is no way to tell where exactly should the stress fall in a word (and it may fall on any syllable, as Bulgarian has free stress). That is why the `Rules` FST should be used here too – it is the only FST, which defines the position of stress. Unfortunately, this means, that such a transcription module will only work with the word forms, produced by the lexicon. An FST defining the relation orthography – phonetics looks as follows:

```
[Rules .o. Ortho].i .o. [Rules .o. Phono]
```

The last system I am going to discuss is a system that relates an arbitrary verb form in Bulgarian to the corresponding 1st person singular non-past tense form of the verb – the verb's dictionary entry. Such a system may be useful in the development of electronic dictionaries – where the user would be able to input an arbitrary verb form and still find the correct dictionary entry.

For the implementation of such a system, one more filter should be written – a filter that changes whatever analysis features are present in the analysis string to the 1st person singular non-past tense analysis features:

```
define F [M+] -> [«NPast» «1P» «Sg»] || \M _ .#.;
```

Then the required FST will be defined as:

```
[Rules .o. Ortho].i .o. F .o. [Rules .o. Ortho]
```

As presented in this section, you will see that there is a very broad range of possible uses of a finite-state implementation of the inflectional morphology of a language. Having defined only a few modules, one can combine them in many different ways to match the specific needs of the project he/she is working on.

3.2. Testing the System

In this section I am discussing the different difficulties I had during the testing of the system and the solutions I came up with.

As it is well known, every system needs to go through a certain amount of testing before it can be pronounced stable and working. In the case of finite-state implementations of linguistic theory, the testing usually amounts in generating all possible forms from the system and then checking their consistence manually. My problem with the testing comes from the fact, that I am using the free version of XFST – which is limited to output not more that 500 forms from a FSA or FST. As one can derive about 35 forms on average from a single base form, my system soon went above that limit. That meant that I have to find some other way for generating all forms from the system.

I noticed that if one calls the built-in XFST function `print random-lower`/`print random-upper` a high enough number of times, it would eventually generate every possible string at least once for the given FST[7]. Following this observation, I implemented a simple Perl script, which prepared an XFST script file and then run it through XFST. The XFST script file contained instructions for loading the system's FST and for issuing the random operations sufficient number of times. The Perl script then collected the output from XFST and, eliminating the duplicates, produced a single list with forms, generated from the system.

Then I went further and extended the script, so that it passes the generated list through the system and collects the resulting analysis data. This resulted in lists of pairs of strings – respectively input and output strings of the system's FST – making the consistency checking easier.

This testing strategy worked well until all problems with the initial set of stems were solved. Then I started adding new stems to the lexicon. At this point, I would rather see only the newly produced forms for checking instead of having to seek for them in the growing list of all forms. Of course, turning off all old stems in the lexicon could do this. But this way I would not notice if a change in the rules affected other stems besides the newly added, or not.

I quickly come up with a solution to this problem. I decided to leave all stems in the lexicon turned on and every time produce all possible forms. Each time I was sure, that all and only correct forms are produced, I would make a backup copy of the list of forms. Then I would add a stem to the lexicon and generate a new list of forms. Afterwards, I would use the `diff` UNIX tool to compare the two lists. In the comparison, then, I would only see the strings that have newly appeared in the list of forms and would be

[7] Actually, this does not work when flags are present. The reason is that the network gets highly unbalanced – with a huge number of paths that do not produce output because of the flags' configuration. As the random operators randomly choose *arcs* while traversing the FST's network and not *entire paths* through the network, they quite often reach dead ends – thus reducing the chance of getting all possible 'correct' paths.

able to easily check their correctness. In addition, the comparison would also contain information for any strings that have disappeared (i.e. that are on the old list, but do not appear on the new one) or have changed. Of course, I also automated this process by adding the necessary code to the Perl script.

As you see, the process of testing an XFST implementation (or any finite-state implementation, I guess) can be made almost automatic and very simple using the power of Perl scripts for manipulating string data.

In this chapter, I presented the technical details of the XFST implementation of the linguistic theory of (Aronson, 1968) for the Bulgarian inflectional morphology.

I showed how the LEXC lexicon is constructed and how it can be further extended. I also showed how (Aronson, 1968)'s morphophonemic alternation rules are implemented using XFST.

After presenting the complete structure of the system, I gave a short overview of the testing methodology I used.

4. Conclusions

In this work, I started with discussing the field of finite-state technology, turning in detail to the uses and advantages of finite-state morphology – showing that with today's computers such systems can have different successful applications in the field of computational linguistics.

Next, I talked about the Bulgarian verbal system and the theory of (Aronson, 1968) that I implemented. I showed the complexity of the inflectional processed that take place and the multitude of verb forms that can be derived from a single stem. I also presented the advantages that this theory has to (Scatton, 1984)'s for the finite-state implementation.

Finally, I presented the technical details of the implementation. I showed what use I made of the XFST tools and the regular expression devices they offer. I explained how several different systems can be derived from this implementation and presented the testing techniques I used.

As a conclusion, I would like to say that this project has brought me a great amount of experience. During the implementation process I got familiar with the XFST and I look forward to extending the system I developed.

I hope that such a system will be useful for XFST learners as well as for language learners. That is why I am planning to make the system available for on-line use until the end of November 2005. In addition, I will make all XFST source files available, as well as the Perl script that I used for testing.

Appendix A Transliteration and Notation Used

Here you will find the transliteration and notation rules I use in my project.

Cyrillic	Latin	IPA	Cyrillic	Latin	IPA
а	a	ɑ	п	p	p
б	b	b	р	r	r
в	v	v	с	s	s
г	g	g	т	t	t
д	d	d	у	u	u
е	e	ɛ	ф	f	f
ж	ž	ʒ	х	x	x
з	z	z	ц	c	ʦ
и	i	i	ч	č	ʧ
й	j	j	ш	š	ʃ
к	k	k	щ	št	-
л	l	l̪	ъ	y	ə
м	m	m	ь	j	ʲ
н	n	n	ю	ju	-
о	o	ɔ	я	ja	-

Table 1 *Transliteration Rules*

In the Latin transcription the stress is marked on the stressed vowel: á, é, í, ó, ú, ý. The # symbol represents the zero vowel found in some stems.

In the Cyrillic part of the system the following symbols are used:

´ → marks the position of the stress – in the final form it means that the following vowel is stressed

° → marks the following vowel as atonic, i.e. not able to bear stress – such vowels are found in some desinences

¶ → represents the zero vowel found in some stems

ø → represents zero desinences

÷ → represents morpheme boundaries

÷÷ → represents the stem-desinence boundary

¬ → marks the following desinence as truncating j → either marks the preceding vowel as palatalized or represents the glide /j/

The markup features used can be divided in two groups – analysis and internal features. The analysis features describe the grammatical form derived and the internal features are used to adjust the application of the rules. The analysis features are the following:

`«NPast»` → non-past tense
`«Aor»` → aorist past tense
`«Imp»` → imperfect past tense
`«Part»` → participial form
`«Pass»` → passive voice
`«Ger»` → gerund form
`«Imper»` → imperative form
`«VSubst»` → verbal substantive
`«1P»`, `«2P»`, `«3P»` → person markers
`«Sg»`, `«Pl»` → number markers
`«m»`, `«f»`, `«n»` → gender markers (*masculine*, *feminine*, *neuter*)

The internal features are the following:

`«Pref»` → prefixed form
`«Trunc»` → truncating stem
`«VS»` → vocalic stem
`«CS»` → consonantal stem
`«Exc1»` → marks the stems with exceptional singular desinence distribution of the imperative forms
`«Exc2»` → marks the stems with exceptional distribution of the non-terminal desinences of the non-past tense
`«Exc3»` → marks the stems with exceptional depalatalizing for the verbal substantive
`«Exc4»` → marks the stems with exceptional stress shifts
`«Exc5»` → marks the stem variant `c´a-` of the verb *be*

Appendix B The LEXC Lexicon

In this appendix I present the source of the LEXC lexicon (file `bop.lexc`).

```
Definitions
      v     = "@D.SecPrefix@" "@R.Prefix.v@";
      vyz   = "@D.SecPrefix@" "@R.Prefix.vyz@";
      do    = "@D.SecPrefix@" "@R.Prefix.do@";
      …
      s     = "@D.SecPrefix@" "@R.Prefix.s@";
      sy    = "@D.SecPrefix@" "@R.Prefix.sy@";
      u     = "@D.SecPrefix@" "@R.Prefix.u@";
      fv    = "@R.Prefix.v@";
      fvyz  = "@R.Prefix.vyz@";
      fdo   = "@R.Prefix.do@";
      …
      fs    = "@R.Prefix.s@";
      fsy   = "@R.Prefix.sy@";
      fu    = "@R.Prefix.u@";
      sv    = "@R.SecPrefix.v@";
      svyz  = "@R.SecPrefix.vyz@";
      sdo   = "@R.SecPrefix.do@";
      …
      ss    = "@R.SecPrefix.s@";
      ssy   = "@R.SecPrefix.sy@";
      su    = "@R.SecPrefix.u@";
      sus   = "@R.SecPrefix.us@";
      dp    = "@D.Prefix@";

      paimp = "@P.Asp.Imp@";
      naimp = "@N.Asp.Imp@";
      raimp = "@R.Asp.Imp@";

      pto   = "@P.Trans.ON@";
      nto   = "@N.Trans.ON@";
      rto   = "@R.Trans.ON@";
```

```
LEXICON Root
<0:«Pref»>                Prefixes;
                          StemTypes;

LEXICON Prefixes
<в "@P.Prefix.v@">        SecPrefixes;
<{въз} "@P.Prefix.vyz@">  SecPrefixes;
<{до} "@P.Prefix.do@">    SecPrefixes;
…
<с "@P.Prefix.s@">        SecPrefixes;
<{съ} "@P.Prefix.sy@">    SecPrefixes;
<у "@P.Prefix.u@">        SecPrefixes;

LEXICON SecPrefixes
<в "@P.SecPrefix.v@">     Stems;
<{въз} "@P.SecPrefix.vyz@"> Stems;
<{до} "@P.SecPrefix.do@"> Stems;
…
<с "@P.SecPrefix.s@">     Stems;
<{съ} "@P.SecPrefix.sy@"> Stems;
<у "@P.SecPrefix.u@">     Stems;
<{ус} "@P.SecPrefix.us@"> Stems;
                          Stems;

LEXICON Stems
0:÷                       StemTypes;

LEXICON StemTypes
<nto naimp>               NTPfStems;
<nto paimp>               NTImpStems;
<pto naimp>               TPfStems;
<pto paimp>               TImpStems;
<nto>                     NTDerStems;
<pto>                     TDerStems;
```

```
LEXICON NTPfStems
<{буј´ај} 0:[«Trunc» «CS»] po>                                                       Des;         !-bujáj- -> become boisterous
<{љј´ез} 0:«CS» [v|za|iz|fna sv|fpro siz]>                                           Des;         !-ljéz- -> go
<{щј´а} 0:[«Trunc» «VS»] dp>                                                         Des;         !štjá- -> want (auxiliary)
…

LEXICON NTImpStems
<{бдј´е} 0:[«Trunc» «VS»] dp>                                                        Des;         !bdjé- -> be awake
<({благо}){ух´ај} 0:[«Trunc» «CS»] dp>                                               Des;         !(blago)uxáj- -> smell sweet
<{буч´а} 0:[«Trunc» «VS»] [[[za|iz]naimp]|dp]>                                       Des;         !bučá- -> rumble
<{б´ъд} 0:«CS» dp>                                                                   Des;         !býd- -> be
<{валј´е} 0:[«Trunc» «VS»] [[[iz|na|fpo sna|fpo spre|[[za|po|pre]0:«Exc3» pto]]naimp]|dp]> Des;   !valjé- -> rain
<{врј´е} 0:[«Trunc» «VS»] [[[vyz|za|iz|na|po|pre|raz|u]naimp]|dp]>                   Des;         !vrjé- -> boil
<{греб´а} 0:[«Trunc» «VS» «Exc4»] [[[[[za|iz|na|po]pto]|o|fpo sna]naimp]|dp]>        Des;         !grebá- -> scoop
<{звънј´е} 0:[«Trunc» «VS»] [[[za|iz|po|pro]naimp]|dp]>                              Des;         !zvynjé- -> ring
<{´ид} 0:«CS» [[[ot|fraz sot]naimp]|dp]>                                             Des;         !íd- -> go
<{´јд} 0:«CS» [do|fpri sdo]naimp>                                                    Des;         !´jd- -> come
<{мир´иса} 0:[«Trunc» «VS»] [[[[v pto]|za|iz|na|po|fpo sna|raz|u]naimp]|dp]>         Des;         !mirísa- -> smell
<{м¶рј´е} 0:[«Trunc» «VS»] [[[za|iz|ot|pre|pri|u]naimp]|dp]>                         Des;         !m#rjé- -> die
<{нјем} [[[za|o]naimp]|dp]>                                                          jej-suff;    !njem-jéj- -> grow mute
<{пас´} 0:«CS» [[[[[iz|o]pto]|na|po]naimp]|dp]>                                      Des;         !pas´- -> graze
<[{раст´} 0:«CS»|{раст´} 0:{÷н°а} 0:[«Trunc» «VS»]] [[[do|za|iz|na|ob|ot|pod|fpo sna|fpo sot|fpo spo|pre|pri   ↩
             |pro|raz|s]naimp]|dp]> Des; !rast´-/rást-na- -> grow
<{с´а} 0:[«VS» «Exc5»] dp>                                                           Des;         !s´a- -> be
<{телефон} dp>                                                                       ira-suff;    !telefon-íra- -> telephone
…

LEXICON TPfStems
<{бљек´} 0:«CS» [o|fpre so|fraz ssy|sy]>                                             Des;         !-bljek´- -> dress
<{валј´и} 0:[«Trunc» «VS»] [za|po|pre|pro|raz|s]>                                    Des;         !-valjí- -> rain (variant)
<{´ем} 0:[«Trunc» «CS»] [za|na|po|pod|fpre sna|pri]>                                 Des;         !-ém- -> take (variant)
<{з´ем} 0:[«Trunc» «CS»] [[v|fza sv|iz|fna sv|nad|fnad sv|ob|fpre sv|fsy sv]|dp]>    Des;         !zém- -> take
<(и){зограф´} dp>                                                                    isa-suff;    !(i)zograf´-isa -> paint
<{н´ем} 0:[«Trunc» «CS»] [ot|fpre ss|s]>                                             Des;         !-ném- -> take (variant)
<{´уј} 0:[«Trunc» «CS»] [iz|ob]>                                                     Des;         !-új- -> put on/off
…
```

```
LEXICON TImpStems
<{б¶р´а} 0:[«Trunc» «VS»] [[[[[po|za|fpo sna|fpo ssy]nto]|do|iz|na|o|ot|pod|fpre siz|pri|pro|raz|s|sy]naimp]|dp]> Des; ↩
                                                                                                               !b#rá- -> take
<{вез´а} 0:[«Trunc» «VS» «Exc4»] dp>                                                 Des;         !vezá- -> embroider
<{гл´еда} 0:«VS» [[[[[do|po|pro]nto]|v|za|iz|na|o|ot|pre|raz|s|sy]naimp]|dp]>        Des;         !gléda- -> watch
<{гл´озга} 0:[«Trunc» «VS»] [[[[po nto]|iz|o]naimp]|dp]>                             Des;         !glózga- -> gnaw
<{др´аска} 0:[«Trunc» «VS»] [[[za|iz|fiz spo|na|o|fpo siz|[po nto]]naimp]|dp]>       Des;         !dráska- -> scratch
<{м´¶лје} 0:[«Trunc» «VS»] [[[do|po|pre|s]naimp]|dp]>                                Des;         !m´#lje- -> mill
<{търпј´е} 0:[«Trunc» «VS»] [[[[[po|s]nto]|iz|pre]naimp]|dp]>                        Des;         !tyrpjé- -> endure
<{ч´ака} 0:«VS» [[[[[za|po]nto]|do|iz|pri]naimp]|dp]>                                Des;         !čáka- -> wait
<{јад´} 0:«CS» [[[do|fdo siz|[[za|nad|o|pod|fpo spre]nto]|iz|fiz spo|na|pre|pro|raz|u]naimp]|dp]> Des; !jad´- -> eat
…

LEXICON NTDerStems
<{блестј} za>                                                                        ava-suff;    !blestj-áva- -> shine
<{б´ъбрј} [iz|na|po|raz]>                                                            a-suff;      !býbrj-a- -> chatter
<{валј} [iz|na|fpo sna|fpo spre|za|po|pre]>                                          ava-suff;    !valj-áva- -> rain
<{интерес} [[[za|fpo sza]pto]|po|dp]>                                                uva-suff;   !interes-úva- -> take interest
…

LEXICON TDerStems
<{б´и} [v|do|za|iz|na|ot|po|pod|[fpo sot nto]|pre|fpri sdo|pro|raz|s|fs sdo|u]>      a-suff;      !-bí-va- -> beat
<{б´ир} [[[po|za|fpo sna|fpo ssy]nto]|do|iz|na|o|ot|pod|fpre siz|pri|pro|raz|s|sy]>  a-suff;      !-bír-a- -> take
<{бл´из} [iz|o|[po nto]|dp]>                                                         va-suff;     !blíz-va- -> lick
<{в´алј} [za|po|pre|pro|raz|s]>                                                      a-suff;      !-valj´-a- -> rain (variant)
<{в´ир} [v|za|na|pro|s]>                                                             a-suff;      !-vír-a- -> shove
<{вонј} [v|za|raz|u]>                                                                ava-suff;    !-vonj-áva- -> stench
<{д´иг} [v|vyz|za|iz|na|po|fpo sv|fpri spo|raz]>                                     na-suff;     !-díg-na -> raise
<{пл´ащ} [[[do|fdo siz|na|raz]nto]|za|iz|nad|ot|pred|dp]>                            a-suff;      !plášt-a- -> pay
<{хладј} [[[za|fpo sza|fpo so|fpo sraz]nto]|o|pro|raz]>                              ava-suff;    !-xladj-áva- -> cool
…

LEXICON jej-suff
<0:÷ {j´ej} 0:[«Trunc» «CS»]>    Des;

LEXICON ira-suff
<0:÷ {´ира} 0:«VS»>              Des;

LEXICON isa-suff
<0:÷ {иса} 0:«VS»>               Des;
```

```
LEXICON uva-suff
<0:÷ {´ува} 0:«VS» paimp>        Des;

LEXICON ava-suff
<0:÷ {´ава} 0:«VS» paimp>        Des;

LEXICON va-suff
<0:÷ {ва} 0:«VS» paimp>          Des;
LEXICON a-suff
<0:÷ a 0:«VS» paimp>             Des;

LEXICON na-suff
<0:÷ {на} 0:[«Trunc» «VS»] naimp> Des;

LEXICON Des
0:÷÷                             DesTypes;

LEXICON DesTypes
<«NPast»>                        NPastNT;
<«Imp»>                          ImperfectNT;
<«Imper» 0:и>                    ImperativeT;
<«Ger» 0:{ejки} raimp>           #;
<«Aor»>                          AoristNT;
<«Pass» «Part» 0:{¬ен÷} rto>     PartT;
<«VSubst» 0:{¬ен÷} «Sg» 0:e raimp> #;

LEXICON NPastNT
<0:{и÷}>                         NPastT;
<«Part» 0:{ещ÷} raimp>           PartT;

LEXICON ImperfectNT
<0:{ex÷}>                        ImperfectT;
<«Part» 0:{ел÷}>                 PartT;
```

```
LEXICON AoristNT
<0:{¬ox÷}>                       AoristT;
<«Part» 0:{л÷}>                  PartT;

LEXICON NPastT
<«1P» «Sg» 0:ъ>                  #;
<«2P» «Sg» 0:ш>                  #;
<«3P» «Sg» 0:ø>                  #;
<«1P» «Pl» 0:м>                  #;
<«2P» «Pl» 0:{т°е}>              #;
<«3P» «Pl» 0:{ът}>               #;

LEXICON ImperfectT
<«1P» «Sg» 0:ø>                  #;
<[«2P»|«3P»] «Sg» 0:e>           #;
<«1P» «Pl» 0:{м°е}>              #;
<«2P» «Pl» 0:{т°е}>              #;
<«3P» «Pl» 0:{°а}>               #;

LEXICON AoristT
<[«1P»|«2P»|«3P»] «Sg» 0:ø>      #;
<«1P» «Pl» 0:{м°е}>              #;
<«2P» «Pl» 0:{т°е}>              #;
<«3P» «Pl» 0:{°а}>               #;

LEXICON PartT
<«m» «Sg» 0:ø>                   #;
<«f» «Sg» 0:a>                   #;
<«n» «Sg» 0:o>                   #;
<«Pl» 0:и>                       #;

LEXICON ImperativeT
<«Sg»>                           #;
<0:{÷т°е} «Pl»>                  #;
```

Appendix C The XFST Implementation

Here I present the sources of the XFST implementation containing the implementations of the morphophonological rules and the script files that generate the system.

The first file is `bop-urules.xfst` – containing a few definitions and an FST that tidies up the input of the system.

```
define AS [«Aor»|«Pass»|«VSubst»];
define NPS [«Imp»|«NPast»|«Imper»|«Ger»];
define M [AS|NPS|
      «Pref»|«Trunc»|«Part»|
      «1P»|«2P»|«3P»|«Sg»|«Pl»|«m»|«f»|«n»|
      «VS»|«CS»|
      «Exc1»|«Exc2»|«Exc3»|«Exc4»|«Exc5»];
define C [b|v|g|d|ž|z|j|k|l|m|n|p|r|s|t|f|x|c|č|š];

define ustress [a -> á, o -> ó, u -> ú, e -> é, i -> í, y -> ý || ´  _, _ ´ C,, ´ -> [] || _ \M+];
```

The next file is `bop-lrules.xfst`. It contains the implementation of the morphophonological alternation rules and some auxiliary FSTs that are used for the derivation of the orthographical and the phonetic representations.

```
define C [б|в|г|д|ж|з|j|к|л|м|н|п|р|с|т|ф|х|ц|ч|ш|щ];
define VelarC [г|к|х];
define HardC [б|в|д|з|м|н|п|р|с|т|ф|ц];
define PalC [ж|ч|ш];
define PalCj [PalC|j];
define VelC [к|г|х];
```

```
define DentC д|з|н|с|т|ц;
define VoiC [б|г|д|з|ж];
define UVoiC [п|ф|к|т|с|ш|х|ц];

define V [а|е|и|о|у|ъ|¶|ю|я];
define AcuteV [е|и];
```

```
define 0PrefixExtention [..] -> ъ || .#. «Pref» в _ ÷ [в|ф|[(´) о]|[п C [V - AcuteV]]];
define 0ConsClustConstr [д|т] -> [] || C _ ÷ DentC;
define 0CjEAlt е -> а || [C - [г|к|ф]] j ´ _ [C - PalCj] [~$[V|j] [V - AcuteV]], [C - [г|к|ф]] j ´ _ х C, ↩
                                                                                   [C - [г|к|ф]] j (´) _ .#.;

define 1AStemTrunc V -> [] || _ «Trunc» «VS» M* ÷ ÷ $NPS ~$¬ .#.,, [j|м] -> [] || _ «Trunc» «CS» M* ÷ ÷ $[AS - «VSubst»];
define 1BDesTrunc V -> [] || .#. ~$«Trunc» «VS» [M - «Exc5»]* ÷ ÷ [M|÷]* (¬) _, ↩
                                «VS» [M - «Exc5»]* ÷ ÷ [[M|÷] - «VSubst»]* ¬ _, «Trunc» «CS» M* ÷ ÷ [[M|÷] - «VSubst»]* ¬ _;
define 1CVelarAlt г -> ж, к -> ч, х -> ш || _ (´) M* ÷ (÷) M* (¬) (´) AcuteV;
define 1DStressAlt [?* ´:0 M* ÷ ÷ [M|÷|C]* (¬) [C|´]* 0:´ V ?*|?* 0:´ V C* ´:0 M* ÷ (÷) [M|÷]* ((¬) C* ° V ?*)| ↩
                                                                 ?* ´:0 ¶ [\V]* 0:´ V ?*|~$[´ ¶ | ´ [÷ [\V]+ ?*|M*] ÷ ÷]];

define 3A1aStemAlt1 [..] -> j || ´ V C* [л|р|п|б|т|м] _ а «Trunc» «VS» $NPS,, ↩
                                       з -> ж, [з[г|д]] -> {ждj}, [с|х] -> ш, {ск} -> {щj} || ´ V C* _ а «Trunc» «VS» $NPS;
define 3A1aStemAlt2 г -> ж, к -> ч || ´ V C* _ а «Trunc» «VS» $NPS;
define 3A1bZeroStemAlt ¶ -> е || _ ~${рj} «Trunc» $NPS,, {рj} -> р, л -> {лj} || ¶ C* _ (´ V) M* ÷ ÷ $NPS;
define 3A2aNTDesFilter и -> е || [«CS»|¶|[[C - PalCj] (´) а «Trunc» «VS»]|[j а «Trunc» «VS»]] ~$«Exc2» «NPast» _ ÷;
define 3A2aNTDesDrop V -> [] || «NPast» _ ÷ M* V [V|C|´|°]* .#.;
define 3A2bTDesExt ъ -> м, м -> {ме} || .#. ~$«Trunc» «VS» $«NPast» «1P» M* _;
define 3B1Palat [..] -> j || HardC _ (´) [«Trunc»|«CS»] $«Imp» ~$[[«2P»|«3P»] «Sg»] .#.;
define 3B2EAlt е -> а || PalCj M* ÷ ÷ [M* & $«Imp»] ´ _ ~$[[«2P»|«3P»] «Sg»] .#.;
define 3C1StressAlt [?* ´:0 ?* «Imper» 0:´ V ?*|~$[«Imper» V]];
define 3C2aSgDesAlt и -> j || .#. [~$«Trunc» & $«VS»] «Imper» _,, и -> [] || .#. $[\C j (´) M* ÷ ÷] «Imper» _, $«Exc1» «Imper» _;
define 3C2cPlDesAlt и -> е || «Imper» _ ÷;
define 3D1Palat [..] -> j || HardC _ (´) [«Trunc»|«CS»] $[«NPast» «Part» (´) а];
define 3D2DesAlt е -> а || ´ M* ÷ ÷ «NPast» «Part»  _;

define 4A1StressAlt [?* 0:´ V C* ´:0 [[M* & $«CS»]|[V ?* «Exc4»]] ÷ ÷ AS ?* .P. ↩
                                                                       ~$[\´ V C* ´ [[M* & $«CS»]|[V ?* «Exc4»]] ÷ ÷ AS]];
define 4A1EAlt е -> а || PalCj ´ _ «Trunc» M* ÷ ÷ [AS - «VSubst»] M* (¬) [\V]* [[V|ø] - и] ?* .#.;
define 4A2HashAlt ø -> е || .#. ~$«Trunc» «CS» $AS [«2P»|«3P»] «Sg» _;
define 4A2RestrictOX {¬ох÷} -> [] || $AS _ [«2P»|«3P»] «Sg»;
define 4B1StemTrunc [д|т] -> [] || _ M* ÷ ÷ [M* & $[«Aor» «Part»]];
define 4B1LAlt [..] -> ъ || [C - [д|т]] (д|т|´) M* ÷ ÷ [M* & $[«Aor» «Part»]] _ л ÷ M* «m»;
define 4C1StemTrunc и -> [] || _ M* ÷ ÷ $«Pass»,, ¬ -> [] || и M* ÷ ÷ ?* «Pass» «Part» _;
define 4C1DePalat j -> [] || C _ ´ «Trunc» $«Pass»;
define 4C2DesSel1 {¬ен} -> т || н (´) а M* ÷ ÷ «Pass» «Part» _, ↩
                          .#. [[$.V & ~$«Pref»]|«Pref» [C|V]* ÷ [$.V & ~$[C {j´ej}]]] «Trunc» «CS» M* ÷ ÷ «Pass» «Part» _;
define 4C2DesSel2 {¬ен} (->) т || .#. [[$.V & ~$«Pref»]|«Pref» [C|V]* ÷ [$.V & $[C {j´ej}]]] «Trunc» «CS» M* ÷ ÷ «Pass» «Part» _;
define 4D1StemTrunc1 и -> [], [н (°) а] -> н, {н´а} -> {н´}, {jа} -> j, {j´а} -> {j´} || _ M* «Trunc» «VS» M* ÷ ÷ $«VSubst»;
define 4D1StemTrunc2 а (->) [] || [VelarC|PalC] ´ _ M* «Trunc» «VS» M* ÷ ÷ $«VSubst»;
define 4D1StemAlt j -> [] || _ ´ V $«Exc3» ÷ ÷ $«Pass»;
define 4D1DesTrunc V -> [] || V «Trunc» «VS» M* ÷ ÷ «VSubst» ¬ _;
define 4D1StressAlt [$¶ ´:0 $«VSubst» [«Sg»|«Pl»] 0:´ ?*|~$[$¶ ´ $«VSubst» [«Sg»|«Pl»] \´]];
define 4D1ZeroStemAlt ¶ -> e || ´ _ $«VSubst»,, V -> [] || $[´ ¶] _ M* $«VSubst»;

define remstress [´|°] -> [];
define repstress [?* 0:´ V ÷ ´:0 j C ?*|~$[´ j C]];
define stress [?* 0:´ j ´:0 [а|е|и|у] ?*|~$[j ´ [а|е|и|у]]];
define finalschwa ъ -> а || _ .#., $«3P» _ т .#.;
define plus ÷ -> [];
define hash [¶|ø] -> [];
define star ¬ -> [];
define mark M -> [];
define pal {jа} -> я, {jу} -> ю, {jе} -> е, {jи} -> и;
define jj j -> й || V _,, j -> ь || C _;
define assim [{дз} -> ц, {дж} -> ч || _ [UVoiC|.#.]] .o. [б -> п, в -> ф, г -> к, д -> т, з -> с, ж -> ш || _ [UVoiC|.#.]] .o. ↩
                                [ц -> {дз}, ч -> {дж} || _ VoiC] .o.  [п -> б, ф -> в, к -> г, т -> д, с -> з, ш -> ж || _ VoiC];
```

```
define Ortho finalschwa .o.
             mark .o.
             repstress .o.
             plus .o.
             hash .o.
             star .o.
             stress .o.
             pal .o.
             jj .o.
             remstress;
```

```
define Phono mark .o.
             repstress .o.
             plus .o.
             hash .o.
             star .o.
             [щ -> [ш т], ° -> []] .o.
             [j -> [] || \VelC _ (´) [е|и]] .o.
             [[..] -> j || VelC _ [е|и]] .o.
             assim .o.
             IPA;
```

The last file I am presenting is the file `b`. This file contains the XFST script, which generates the main system FST.

```
source bop-translit.xfst
source bop-ipa.xfst
source bop-urules.xfst
source bop-lrules.xfst
read lexc bop.lexc
define Rules [ustress.i .o.
      Trans.i .o.
      Lexicon .o.
```

```
eliminate flag Prefix;
eliminate flag SecPrefix;
eliminate flag Asp;
eliminate flag Trans;
define Lexicon;
```

```
[{държ´а} -> [{дръж´} «Exc1»] || _ M+ ÷ ÷ $«Imper»] .o.                              !dyrží -> drýž
[{јад´} -> [{јаж´} «Exc1»] || _ M+ ÷ ÷ $«Imper»] .o.                                 !jadí -> jáž
[{видј´е} -> [{виж´} «Exc1»] || _ M+ ÷ ÷ $«Imper»] .o.                               !vidí -> víž
[{видј´е} -> {в´идје} || _ M+ ÷ ÷ $«NPast»] .o.                                      !vidjá -> vídja
[[..] -> «Exc1» || {÷лј´ез} _ M+ ÷ ÷ $«Imper»] .o.                                   !vlezí -> vléz
[[«Trunc» -> [] || ´ ?* [{зг}|{ск}] а _ «VS» $NPS]|[[..] -> «Exc2» || ´ ?* [{зг}|{ск}] а «Trunc» «VS» _ $NPS]] .o. ↵
                                                                               ![zg|sk]«Trunc» -> [«NTrunc»|«Exc2»]
[а -> и || {душ´} _ M+ ÷ ÷ $[«Pass»|«VSubst»]] .o.                                   !dušá -> duší
[[[{кълн´а} «Trunc»] -> {кл´е} || _ $«Aor»]|[~$[{кълн´а} ?* «Aor» «Part»]]] .o.      !kylnáx -> kléx
[о (->) [] || [[\C ÷]|.#.] з _ {в´а} M+ ÷ ÷ $«Pass»] .o.                             !zová (->) zvá
[е -> а || {рј´} _ {за} M+ ÷ ÷ $AS] .o.                                              !rjéza -> rjáza
[[{ад´} «CS»] -> [{´а} «VS»] || [.#.|÷] [д|ј] _ [$«NPast» & $[«1P» «Sg»]]] .o.       ![d|j]adá -> [d|j]ám
[[{ад´} «CS»] -> [{´а} «VS»] || [.#.|÷] д _ $«Imper»] .o.                            !dadí -> dáj
[[ј «Trunc» «CS»] (->) «VS» || .#. {зн´а} _ [$«NPast» & $[«1P» «Sg»]]] .o.           !znája (->) znám
[[..] -> ј || [.#.|÷] {сп} _ {´а} M+ ÷ ÷ $NPS] .o.                                   !spá -> spjá
[ј -> [] || .#. щ _ {´а} M+ ÷ ÷ $«NPast»] .o.                                        !štjá -> štá
[{к´олји} (->) {к¶л´а} || _ M+ ÷ ÷ $AS] .o.                                          !kólix (->) kláx
[{кълн´а} (->) {к¶л´е} || _ M+ ÷ ÷ $«Aor»] .o.                                       !kylná (->) kléx
[{п´орји} (->) {п¶р´а} || [{от}|{раз}] ÷ _ M+ ÷ ÷ $AS] .o.                           !-pórix (->) -práx
[[{м´ожа} «Trunc» «VS»] -> [{м´ог} «CS»] || _ $«NPast»] .o.                          !móžam -> móga
[[{м´ожа} «Trunc» «VS»] (->) [{мог´} «CS»] || _ $[«Aor» «Part»]] .o.                 !móžel (->) mogýl
[{б´ъд} -> {бид´} || .#. _ M+ ÷ ÷ $[«Aor»|«Ger»]] .o.                                !býl -> bíl
[{б´ъд} (->) {бј´} || .#. _ M+ ÷ ÷ [$«Imp» & ~$«Part»] .#.] .o.                      !býdeše (->) béše
[{б´ъд} -> [?-?] || .#. _ M+ ÷ ÷ $«VSubst»] .o.                                      !býdene -> ø
[{ех} (->) [] || .#. б ј ´ $[÷ ÷ «Imp»] _ $[[«2P»|«3P»]«Sg»]] .o.                    !béše (->) bé
[{с´а} -> [?-?] || .#. _ M+ ÷ ÷ [~$«NPast»|$«Part»] .#.] .o.                         !sá- has only NPast forms
[а -> ъ || .#. с ´ _ M+ ÷ ÷ $[«1P» «Sg»]] .o.                                        !sým
[[а «VS»] -> [«VS»] || .#. {с´} _ $[[«1P»|«2P»]«Pl»|[«2P»|«3P»]«Sg»]] .o.            !smé
[и -> [] || .#. {с´} (а) M+ ÷ ÷ M* _ $[«Pl»|«3P»]] .o.                               !sí
[?* -> [] || .#. {с´} (а) M+ ÷ ÷ M* $[«2P» «Sg»|«3P» «Pl»] _] .o.                    !sá
[° -> [] || .#. {с´} M+ ÷ ÷ $[«2P» «Pl»] т _ е] .o.                                  !sté
[{с´} -> {´е} || .#. _ M+ ÷ ÷ $[«3P» «Sg»]] .o.                                      !é
[{´ид} -> [?-?] || .#. ~$«Pref» _ M+ ÷ ÷ $[«Aor» «Part»]] .o.                        !íd- has no non-Pref Aor Part
[{´ид} -> {´иш} || $«Pref» _ M+ ÷ ÷ $[«Aor» «Part»]] .o.                             !-íd- -> -íš- in Pref Aor Part
[{до÷´јд} -> [{ел´а} «Exc1»] || «Pref» _ M+ ÷ ÷ $«Imper»] .o.                        !dojdí -> elá
[{´јд} -> {јд´} || $«Pref» _ M+ ÷ ÷ $«Aor»] .o.                                      !dójdox -> dojdóx
[{јд´} -> {ш´} || $«Pref» _ M+ ÷ ÷ $[«Aor» «Part»]] .o.                              !-´jd- -> -š´- in Pref Aor Part
0PrefixExtention .o.
```

```
      3A1bZeroStemAlt .o.
      3A2aNTDesFilter .o.
      3A1aStemAlt1 .o.
      3A1aStemAlt2 .o.
      4D1ZeroStemAlt .o.
      4C2DesSel1 .o.
      4C2DesSel2 .o.
      4C1StemTrunc .o.
      4D1StemTrunc1 .o.
      4D1StemTrunc2 .o.
      4D1DesTrunc .o.
      1AStemTrunc .o.
      3A2aNTDesDrop .o.
      3A2bTDesExt .o.
      3C2aSgDesAlt .o.
      3C2cPlDesAlt .o.
      3C1StressAlt .o.
      4A2RestrictOX .o.
      4A2HashAlt .o.
      1CVelarAlt .o.
      3D2DesAlt .o.
      1BDesTrunc .o.
      3B1Palat .o.
      3D1Palat .o.
      4C1DePalat .o.
      4A1StressAlt .o.
      4D1StemAlt .o.
      4D1StressAlt .o.
      4B1LAlt .o.
      1DStressAlt .o.
      4A1EAlt .o.
      4B1StemTrunc .o.
      3B2EAlt .o.
      0CjEAlt .o.
      0ConsClustConstr];

read regex Rules .o. [Phono|Ortho];
```

Appendix D Example Derivations

In this appendix, I will go through the stages of the analysis of a few verb forms, showing the operation of the rules.

The first verb form I am going to look at is доберeше – past Imperfect, 2nd or 3rd person singular from “reach” – in its orthographical representation. I will list only the rules that affect this formation.

```
dob#rá«Imp»«2P»«Sg»
dob#rá«Imp»«3P»«Sg»
↿⇂ ustress.i
dob#r´a«Imp»«2P»«Sg»
dob#r´a«Imp»«3P»«Sg»
↿⇂ Trans.i
доб¶р´а«Imp»«2P»«Sg»
доб¶р´а«Imp»«3P»«Sg»
↿⇂ Lexicon
«Pref»до÷б¶р´а«Trunc»«VS»÷÷«Imp»ех÷«2P»«Sg»е
«Pref»до÷б¶р´а«Trunc»«VS»÷÷«Imp»ех÷«3P»«Sg»е
↿⇂ 3A1bZeroStemAlt
«Pref»до÷бер´а«Trunc»«VS»÷÷«Imp»ех÷«2P»«Sg»е
«Pref»до÷бер´а«Trunc»«VS»÷÷«Imp»ех÷«3P»«Sg»е
↿⇂ 1AStemTrunc
«Pref»до÷бер´«Trunc»«VS»÷÷«Imp»ех÷«2P»«Sg»е
«Pref»до÷бер´«Trunc»«VS»÷÷«Imp»ех÷«3P»«Sg»е
↿⇂ 1CVelarAlt
«Pref»до÷бер´«Trunc»«VS»÷÷«Imp»еш÷«2P»«Sg»е
«Pref»до÷бер´«Trunc»«VS»÷÷«Imp»еш÷«3P»«Sg»е
↿⇂ 3B1Palat
«Pref»до÷берj´«Trunc»«VS»÷÷«Imp»еш÷«2P»«Sg»е
«Pref»до÷берj´«Trunc»«VS»÷÷«Imp»еш÷«3P»«Sg»е
↿⇂ 1DStressAlt
«Pref»до÷берj«Trunc»«VS»÷÷«Imp»´еш÷«2P»«Sg»е
«Pref»до÷берj«Trunc»«VS»÷÷«Imp»´еш÷«3P»«Sg»е
↿⇂ mark
до÷берj÷÷´еш÷е
↿⇂ plus
доберj´еше
↿⇂ stress
добер´jеше
↿⇂ pal
добер´еше
↿⇂ remstress
добереше
```

The second verb form I am going to look at is isʧˈɑkvɑxtɛ – past Aorist or past Imperfect, 2nd person plural from “await” – in its phonetic representation. I will again list only the rules that affect this formation.

```
izčákva«Aor»«2P»«Pl»
izčákva«Imp»«2P»«Pl»
↿⇂ ustress.i
```

```
         izč´akva«Aor»«2P»«Pl»
         izč´akva«Imp»«2P»«Pl»
              ⇅ Trans.i
         изч´аква«Aor»«2P»«Pl»
         изч´аква«Imp»«2P»«Pl»
              ⇅ Lexicon
«Pref»из÷ч´ак÷ва«VS»÷÷«Aor»¬ох÷«2P»«Pl»т°е
«Pref»из÷ч´ак÷ва«VS»÷÷«Imp»ех÷«3P»«Pl»т°е
              ⇅ 1BDesTrunc
«Pref»из÷ч´ак÷ва«VS»÷÷«Aor»¬х÷«2P»«Pl»т°е
«Pref»из÷ч´ак÷ва«VS»÷÷«Imp»х÷«3P»«Pl»т°е
              ⇅ mark
          из÷ч´ак÷ва÷÷¬х÷т°е
           из÷ч´ак÷ва÷÷х÷т°е
              ⇅ plus
             изч´аква¬хт°е
             изч´аквахт°е
              ⇅ star
             изч´аквахт°е
              ⇅ …
             изч´аквахте
              ⇅ assim
             исч´аквахте
              ⇅ IPA
             isʧˈɑkvɑxtɛ
```

As you can see from these derivations, often a single surface form – be it orthographical or phonetic – corresponds to two or more analysis forms.

Next I include lists with the forms that could be derived from different stems. The first stem is `izlj´ez` – a non-transitive perfective stem, meaning *exit*.

```
izljéz«Aor»«1P»«Pl»
излязохме

izljéz«Aor»«1P»«Sg»
излязох

izljéz«Aor»«2P»«Pl»
излязохте

izljéz«Aor»«2P»«Sg»
излезе

izljéz«Aor»«3P»«Pl»
излязоха

izljéz«Aor»«3P»«Sg»
излезе

izljéz«Aor»«Part»«Pl»
излезли

izljéz«Aor»«Part»«f»«Sg»
излязла

izljéz«Aor»«Part»«m»«Sg»
излязъл

izljéz«Aor»«Part»«n»«Sg»
излязло

izljéz«Imper»«Pl»
излезте

izljéz«Imper»«Sg»
излез

izljéz«Imp»«1P»«Pl»
излезехме

izljéz«Imp»«1P»«Sg»
излезех

izljéz«Imp»«2P»«Pl»
излезехте

izljéz«Imp»«2P»«Sg»
излезеше
```

izljéz«Imp»«3P»«Pl»
излезеха

izljéz«Imp»«3P»«Sg»
излезеше

izljéz«Imp»«Part»«Pl»
излезели

izljéz«Imp»«Part»«f»«Sg»
излезела

izljéz«Imp»«Part»«m»«Sg»
излезел

izljéz«Imp»«Part»«n»«Sg»
излезело

izljéz«NPast»«1P»«Pl»
излезем

izljéz«NPast»«1P»«Sg»
изляза

izljéz«NPast»«2P»«Pl»
излезете

izljéz«NPast»«2P»«Sg»
излезеш

izljéz«NPast»«3P»«Pl»
излязат

izljéz«NPast»«3P»«Sg»
излезе

The next stem is `mir´isa` – a non-transitive imperfective stem, meaning *smell*.

mirísa«Aor»«1P»«Pl»
мирисахме

mirísa«Aor»«1P»«Sg»
мирисах

mirísa«Aor»«2P»«Pl»
мирисахте

mirísa«Aor»«2P»«Sg»
мириса

mirísa«Aor»«3P»«Pl»
мирисаха

mirísa«Aor»«3P»«Sg»
мириса

mirísa«Aor»«Part»«Pl»
мирисали

mirísa«Aor»«Part»«f»«Sg»
мирисала

mirísa«Aor»«Part»«m»«Sg»
мирисал

mirísa«Aor»«Part»«n»«Sg»
мирисало

mirísa«Ger»
миришейки

mirísa«Imper»«Pl»
миришете

mirísa«Imper»«Sg»
мириши

mirísa«Imp»«1P»«Pl»
миришехме

mirísa«Imp»«1P»«Sg»
миришех

mirísa«Imp»«2P»«Pl»
миришехте

mirísa«Imp»«2P»«Sg»
миришеше

mirísa«Imp»«3P»«Pl»
миришеха

mirísa«Imp»«3P»«Sg»
миришеше

mirísa«Imp»«Part»«Pl»
миришели

mirísa«Imp»«Part»«f»«Sg»
миришела

mirísa«Imp»«Part»«m»«Sg»
миришел

mirísa«Imp»«Part»«n»«Sg»
миришело

mirísa«NPast»«1P»«Pl»
миришем

mirísa«NPast»«1P»«Sg»
мириша

mirísa«NPast»«2P»«Pl»
миришете

```
mirísa«NPast»«2P»«Sg»
миришеш

mirísa«NPast»«3P»«Pl»
миришат

mirísa«NPast»«3P»«Sg»
мирише

mirísa«NPast»«Part»«Pl»
миришещи

mirísa«NPast»«Part»«f»«Sg»
миришеща

mirísa«NPast»«Part»«m»«Sg»
миришещ

mirísa«NPast»«Part»«n»«Sg»
миришещо

mirísa«VSubst»«Sg»
мирисане
```

The third stem is `preobljek´` – a transitive perfective stem, meaning *redress*.

```
preobljek´«Aor»«1P»«Pl»
преоблякохме

preobljek´«Aor»«1P»«Sg»
преоблякох

preobljek´«Aor»«2P»«Pl»
преоблякохте

preobljek´«Aor»«2P»«Sg»
преоблече

preobljek´«Aor»«3P»«Pl»
преоблякоха

preobljek´«Aor»«3P»«Sg»
преоблече

preobljek´«Aor»«Part»«Pl»
преоблекли

preobljek´«Aor»«Part»«f»«Sg»
преоблякла

preobljek´«Aor»«Part»«m»«Sg»
преоблякъл

preobljek´«Aor»«Part»«n»«Sg»
преоблякло

preobljek´«Imper»«Pl»
преоблечете

preobljek´«Imper»«Sg»
преоблечи

preobljek´«Imp»«1P»«Pl»
преоблечахме

preobljek´«Imp»«1P»«Sg»
преоблечах

preobljek´«Imp»«2P»«Pl»
преоблечахте

preobljek´«Imp»«2P»«Sg»
преоблечеше

preobljek´«Imp»«3P»«Pl»
преоблечаха

preobljek´«Imp»«3P»«Sg»
преоблечеше

preobljek´«Imp»«Part»«Pl»
преоблечали

preobljek´«Imp»«Part»«f»«Sg»
преоблечала

preobljek´«Imp»«Part»«m»«Sg»
преоблечал

preobljek´«Imp»«Part»«n»«Sg»
преоблечало

preobljek´«NPast»«1P»«Pl»
преоблечем

preobljek´«NPast»«1P»«Sg»
преоблека

preobljek´«NPast»«2P»«Pl»
преоблечете

preobljek´«NPast»«2P»«Sg»
преоблечеш

preobljek´«NPast»«3P»«Pl»
преоблекат

preobljek´«NPast»«3P»«Sg»
преоблече

preobljek´«Pass»«Part»«Pl»
преоблечени

preobljek´«Pass»«Part»«f»«Sg»
преоблечена
```

```
preobljek´«Pass»«Part»«m»«Sg»
преоблечен

preobljek´«Pass»«Part»«n»«Sg»
преоблечено
```

The last stem is `dr´aska` – a transitive imperfective stem, meaning *scratch*.

```
dráska«Aor»«1P»«Pl»
драскахме

dráska«Aor»«1P»«Sg»
драсках

dráska«Aor»«2P»«Pl»
драскахте

dráska«Aor»«2P»«Sg»
драска

dráska«Aor»«3P»«Pl»
драскаха

dráska«Aor»«3P»«Sg»
драска

dráska«Aor»«Part»«Pl»
драскали

dráska«Aor»«Part»«f»«Sg»
драскала

dráska«Aor»«Part»«m»«Sg»
драскал

dráska«Aor»«Part»«n»«Sg»
драскало

dráska«Ger»
драскайки
дращейки

dráska«Imper»«Pl»
драскайте
дращете

dráska«Imper»«Sg»
драскай
дращи

dráska«Imp»«1P»«Pl»
драскахме
дращехме

dráska«Imp»«1P»«Sg»
драсках
дращех

dráska«Imp»«2P»«Pl»
драскахте
дращехте

dráska«Imp»«2P»«Sg»
драскаше
дращеше

dráska«Imp»«3P»«Pl»
драскаха
дращеха

dráska«Imp»«3P»«Sg»
драскаше
дращеше

dráska«Imp»«Part»«Pl»
драскали
дращели

dráska«Imp»«Part»«f»«Sg»
драскала
дращела

dráska«Imp»«Part»«m»«Sg»
драскал
дращел

dráska«Imp»«Part»«n»«Sg»
драскало
дращело

dráska«NPast»«1P»«Pl»
драскаме
дращим

dráska«NPast»«1P»«Sg»
драскам
дращя

dráska«NPast»«2P»«Pl»
драскате
дращите

dráska«NPast»«2P»«Sg»
драскаш
дращиш

dráska«NPast»«3P»«Pl»
драскат
дращят

dráska«NPast»«3P»«Sg»
драска
дращи
```

```
dráska«NPast»«Part»«Pl»
драскащи
дращещи

dráska«NPast»«Part»«f»«Sg»
драскаща
дращеща

dráska«NPast»«Part»«m»«Sg»
драскащ
дращещ

dráska«NPast»«Part»«n»«Sg»
драскащо
дращещо
```

```
dráska«Pass»«Part»«Pl»
драскани

dráska«Pass»«Part»«f»«Sg»
драскана

dráska«Pass»«Part»«m»«Sg»
драскан

dráska«Pass»«Part»«n»«Sg»
драскано

dráska«VSubst»«Sg»
драскане
```

Bibliography

Angelov, Stefan. 2005. SA Dictionary 2005 T1. Bulgarian-English and English-Bulgarian Dictionary. http://sa.dir.bg, last accessed 29.07.2005

Aronson, Howard I. 1968. Bulgarian Inflectional Morphology: Slavistic Printings and Reprintings. The Hague: Mouton.

Beesley, Kenneth R. and Lauri Karttunen. 2003. Finite State Morphology. Stanford: CLSI Publications.

Kaplan, R. M. and M. Kay. 1994. Regular Models of Phonological Rule Systems. Computational Linguistics, vol. 20(3): 331-378.

Koskenniemi, Kimmo. 1983. Two-Level Morphology: A General Computational Model for Word-Form Recognition and Production, University of Helsinki: Ph.D. Thesis.

Pashov, Petar and Hristo Parvev. 1979. Pravogovoren Rechnik na Balgarskija Ezik. Sofia: Nauka i Izkustvo.

Scatton, Earnest A. 1984. A Reference Grammar of Modern Bulgarian. Columbus, Ohio: Slavica Publishers.