The use of Extract Morphology for Automatic Generation of Language Technology for Votic

Kristian Kankainen
University of Tartu
Institute of Estonian and General Linguistics
kristian@keeleleek.ee

Abstract

The article presents a source code generating extension to Språkbanken's morphological dictionary building tool, the "Morphology lab". This is done for three reasons: 1) to include the speech community in the morphological dictionary making 2) to enable a time-resistant, human readable description of Votic morphology 3) source code generation minimizes the efforts keeping multiple language technologies in sync when the morphological dictionary is updated.

The morphological dictionary tool uses Extract Morphology that extracts the paradigm descriptions automatically from user input inflection tables. In this way the user's linguistic and technological knowledge is kept at a bare minimum.

The presented extension encodes both the lexical information and the procedural paradigm descriptions into the ISO standard Lexical Markup Framework (LMF). From this central representation program source code is generated for morphological analysis and synthesis modules in two language technological systems: the Grammatical Framework and the Giellatekno infrastructure.

Integration into the Giellatekno infrastructure is still work in progress, but the extension has allready been successfully used as a continuous integration platform for developing the morphology module of the Votic Resource Grammar Library in the Grammatical Framework.

In the end the article discusses four main implications of the presented approach: 1) the work on morphology is reduced to an interface similar to Wiktionary, 2) the lexical resource is put in the middle, 3) the general benefits of source code generation, 4) benefits of lemma form agnosticity inherent in the approach.

Teesid

Artikkel tutvustab koodigenereerimise laiendust Språkbankeni morfoloogiliste sõnaraamatute koostamissüsteemile. Laiendusel on kolm eesmärki: 1) kaasamaks kõnelejaskonda morfoloogilise sõnastiku koostamisprotsessi; 2) võimaldada arhiveeritava ja inimloetava morfoloogiakirjelduse koostamise; 3) koodigenereerimine minimeerib jõupingutused, et hoida mitut rööpset keeletehnoloogiat ajakohastena morfoloogilise sõnaraamatu uuendamise puhul.

Morfoloogilise sõnaraamatu koostamissüsteem põhineb ekstraktmorfoloogial, mis eraldab automaatselt paradigmakirjelduse tüüpsõna muutvormitabelist. Sellisel viisil viiakse kasutaja vajatud keelelised ja tehnoloogilised teadmised miinimumi.

This work is licensed under a Creative Commons Attribution 4.0 International Licence. Licence details:
http://creativecommons.org/licenses/by/4.0/

Proceedings of the fifth Workshop on Computational Linguistics for Uralic Languages, pages 192–204,
Tartu, Estonia, January 7 - January 8, 2019.

Tutvustatud laiendus teisendab nii leksikaalse informatsiooni, kui ka paradigmade arvutuslikud kirjeldused rahvusvahelise standardi kujule ISO Lexical Markup Framework. Sellest kesksest kujust genereeritakse programmkood kahe keeletehnoloogilise raamistiku morfoloogiamoodulite jaoks: programmeerimiskeel Grammatical Framework ja Giellatekno infrastruktuur.

Integreerimine Giellatekno infrastruktuuri on pooleli. Ent tutvustatud laiendust on edukalt kasutatud vadja keele Grammatical Framework'i morfoloogiamooduli genereerimiseks.

Artikli lõpus käsitletakse neli teemat, mida tutvustatud lähenemine toob kaasa: 1) töö morfoloogilise kirjelduse kallal on taandatud muutvormide tabelite esitamisele, 2) töö keskseks osaks on leksikaalse ressursi loomine, 3) koodigenereerimise üldised hüved, ning 4) tutvustatud lähenemise lemmavormi valimise sõltumatuse hüved.

1 Introduction

The presented system is part of a broader work on defining a normative description of Votic morphology to be used in corpus planning, language teaching and language revitalization efforts.

There was no literature developed for Votic in the 1930s (unlike for Karelian, Veps, Ingrian and other languages in the Soviet Union). But still the Votic language has many linguistic descriptions, including grammars (e.g. Ahlqvist (1856), Tsvetkov (2008), Ariste (1968), Маркус and Рожанский (2011)) and big (electronic) dictionaries (Tsvetkov (1995), Grünberg et al. (2013)) but very few that gives definitive answers to language learners asking *how is lexeme XYZ in genitive form?* For this a standardization or normative description of the morphology is needed. Metaphorically, the presented system tries to give the fishing rod instead of only the fish. That is, to give to the language community not only the normative morphology but a system where the normative can be altered. This is seen as crucial for the group of less than 20 active Votic language activists.

To minimize efforts, a morphological dictionary is compiled in such a way that it can be used both as a reference guide of the normative morphological description and as a basis for automatically generating the source codes implementing the morphological analysis and synthesis modules for two language technological platforms: the Giellatekno infrastructure and the Grammatical Framework.

Planned future applications for the generated morphological modules is proofing tools obtained from the integration with Giellatekno's infrastructure, and a bilingual Russian–Votic phrasebook application done in the Grammatical Framework.

The morphological dictionary is built with an existing tool called the "Morphology Lab". The presented method is not language dependent, but so far work has only been done with the Votic language. The most language dependent part of the system is the type system for the generated Grammatical Framework source code and this concerns only the optimization of computer memory usage.

The system has been successfully used for continuous integration when working on the morphology module of the Votic Resource Grammar Library in the Grammatical Framework. New paradigm functions has continuously and automatically been added to the morphology module when the corresponding words' inflection tables have been added to the dictionary through the Morphology Lab.

Initial work for generating source code and integrating with the Giellatekno infrastructure is presented. This is work in progress that has not been deployed nor

fully tested.

The article first presents the Morphology Lab tool's workflow for working on the Votic morphological dictionary and how it uses Extract Morphology for paradigm extraction. Section 3 introduces source code generation and section 3.1 how the morphological dictionary's data is used as its ontology represented in the Lexical Markup Framework. Section 3.2 explains the source code generation for GF and section 3.3 for Giellatekno. Lastly, section 4 discusses the advantage and main implications of using the presented approach. Generated source code examples are added at the end.

2 Workflow: Morphology Lab and Extract Morphology

The Morphology Lab is a separate online tool for working with morphological dictionaries. It was created as a prototype to integrate Extract Morphology into Språkbanken's lexical infrastructure Karp. It uses Extract Morphology to predict correct paradigms for new lexical entries, and to extract new paradigms not yet existing in the system from user input example inflection tables. It also integrates with the corpus tool Korp and shows frequency counts for wordforms in the inflection tables.

The workflow for building the Votic morphological dictionary with the Morphology Lab has been straight forward. Although no users from the Votic speech community has yet been found, the tool is seen simple enough to encourage their inclusion.

For a new entry, the user first inserts the new word (or any of its wordforms) and chooses the correct paradigm(s) for it. The paradigm can be specified either by inserting another word that shares the same paradigm or by reviewing the generated inflection tables and choosing the appropriate one(s). The generated inflection tables show for each wordform also their frequency count in a corpus.

If a correct paradigm does not yet exist in the system, the user has to input all wordforms in the inflection table which is then saved as a new paradigm. The user can do this by filling a blank inflection table or by editing wordforms in a generated (wrong) inflection table. The newly saved paradigm is directly reflected in the system and can be chosen when adding the next new entry to the system.

Next we will introduce Extract Morphology. The paradigm extraction facility of Extract Morphology has been proposed as a more natural way for a linguist to define a computational morphology (Forsberg (2016)) as it doesn't rely on any specific knowledge other than wordforms. It has been presented in detail in (Ahlberg et al. (2014)) and (Ahlberg et al. (2015)). Extract Morphology has been used for deriving morphological guessers as weighted and unweighted finite state transducers (Forsberg and Hulden (2016)). The design of its extraction techniques in finite state calculus has been presented more thoroughly in (Hulden (2014)).

At the core, Extract Morphology's paradigm extraction finds the longest common subsequence (LCS) across all wordforms in an inflection table. These common parts are used as a technical stem for defining the lexeme. The common and non-common parts make up concatenation patterns for each wordform.

As a simplified example, consider the two wordforms *tšiutto* and *tšiutod* ('shirt'). Their longest common subsequences are highlighted in table 1.

The common parts are abstracted away into variables and thus define concatenation templates for the paradigm's wordforms, such as: $x_1 \oplus \mathbf{t} \oplus x_2$ and $x_1 \oplus x_2 \oplus \mathbf{d}$.

It is obvious that replacing the variables with the values x_1=**tšiut** and x_2=**o** recreates the inital inflection table. Furthermore, other words sharing the same paradigm

Wordform	MSD	LCS
tšiutto	Sg Nom	tšiut t o
tšiutod	Pl Nom	tšiut o d

Table 1: Inflection table with morphosyntactic descriptions and underlined longest common subsequence.

can also be instantiated, for example x_1=**kat** and x_2=**o** instantiates the lexeme's *katto* paradigm.

The technical stem creates a too low-level interface for any practical human use. But by choosing one wordform to be used as lemma, the concatenation pattern can be used to construct a dispatch function that segment the complete lemma wordform into the needed technical stem parts. This approach is illustrated in the Grammatical Framework source code generation in section 3.2.

Since the technical stem is independent of any wordform in the paradigm, it is also independent from the choice of what is used as the dictionary lemma.

This has several implications. The choice of lemma form in the dictionary meant for human consumption can be post-poned after analysing the principal parts of the paradigms or consulting the end dictionary users. Also, the choice of lemma form used in the generated language technology need not be the same and can be chosen in accordance to specific needs of each generated software individually. Ideally though, the same form is used in all systems, but the point here is that this choice can be made on-the-go, or when enough data has been collected.

This kind of lemma agnosticity in the work flow is seen as a good thing for languages that lack a lexicographic tradition and is discussed more closely in section 4.

3 Source code generation

Source code generation is a loose term used here to designate the automatic process where program source code is generated from a description or ontology. The ontology describes the *what*, not the *how* of the morphological knowledge, whereas the generators transform this knowledge into a program implementation. The implementation is a derived kind of *how*, trying as close as possible to resemble the original *what* of the ontology.

In our work we represent the ontology using the ISO standard Lexical Markup Framework (LMF). All the data recorded by the Morphology Lab is converted into LMF form and this is used as input for the source code generator.

Two source code generators are presented below, one for the Grammatical Framework and one for the Giellatekno infrastructure. The source code generators are implemented with the XQuery programming language.

A shared architectural feature of both code generators is the use of translation tables for translating terms used in the LMF ontology to their corresponding terms used in the host environment. For example the names of paradigms are prefixed with `as` in the LMF (e.g. `asTšiutto`), but named like actions in the GF (`mkTšiutto`), and prefixed by their part of speech in Giellatekno (`N_TŠIUTTO`). Also the terminology for grammatical features differ between the environments.

In this way different terminological traditions are supported and respected.

3.1 Lexical Markup Framework

The Lexical Markup Framework (LMF, ISO 24613:2008) is an ISO standard for natural language processing lexicons and machine readable dictionaries. It provides a common model for managing exchange of data and enables merging of different resources. (Francopoulo, 2013).

The LMF standard consists of a core model and several extensions. In our work we use two extensions: the *NLP Morphological Pattern* extension to model the extracted paradigm information, and the *Morphology* extension to represent each lexeme's inflected wordforms.

Representing the lexeme's morphology with both paradigms (describing in intension) and inflected wordform tables (describing in extension) might seem superfluous. But both representations serve their own purpose.

Listing extensionally all inflectional wordforms for each lexeme is in this work considered part of documentation and what adds value to the work's 50-year perspective (discussed in section 4).

Recording lexeme's all inflected wordforms extensionally also creates the possibility to further annotate the individual wordforms, such as real attested corpus attestations, or other meta-linguistic information such as judgements.

Listing wordform information explicitly is also beneficial for the dictionary system, enabling quick searches and statistics.

Next, we will introduce our data and how it is represented in the LMF.

3.1.1 Representing the lexical entries

All lexical entries carry information about their part of speech, their paradigm(s), the (chosen) lemma form, and the inflected wordforms together with their morphosyntactic descriptors. Also some additional information specific to Karp and the Morpholoy Lab is stored.

The lexical entry for our simplified example is shown in figure 1.

3.1.2 Representing the paradigms

The extracted paradigms are represented as LMF Morphological Patterns. These hold information about their part of speech and are name-tagged with an ID. The names follow the LMF tradition and are prefixed with `as`, such as `asTšiutto`.

The LMF Morphological Patterns model all the information extracted by Extract Morphology. The attested variable values, i.e. the technical stems, of all lexemes added in the Morphology Lab is saved. This information could be utilized to integrate prediction models into the generated source code, as have been demonstrated by Forsberg and Hulden (2016). This has not been done, as the work has focused on integrating the lexical resources as a first stage.

The extracted paradigm's concatenation patterns are represented as LMF Transform Sets. These hold, for each inflected wordform, the morphosyntactic description and an ordered list of Processes which model the concatenation of constant strings (e.g. the non-common parts) and instantiable variables (e.g. the common parts, the LCS).

Furthermore, the full LMF Lexical Resource also holds global meta information such as language name and language code. This is used mainly to name the generated source code files.

Thus, for our simplified example we get its extracted paradigm represented as shown in figure 2.

3.2 Generating Source Code for the Grammatical Framework

Grammatical Framework (GF) is a special purpose programming language for grammars (Ranta, 2011). It can treat grammars as software libraries called Resource Grammar Libraries (Ranta, 2008) and the aim of the work presented here is to generate the source code that implements the morphology module for the Votic resource grammar library.

GF's concept of a paradigm is similar to that of Extract Morphology, and writing the source code generator has been intuitive.

3.2.1 Generating the lexicon

GF is a multilingual framework and each application is expected to define their own semantics in an abstract grammar. No attempt is made to include semantic pivots to the Votic resource, instead a simple monolingual word list is generated as a lexicon.

This lexicon specifies for each entry in the resource its lemma, which is appended with its part of speech and a call to the paradigm function. In the case a lemma has multiple paradigms, each one is declared as variants.

The GF developer could use this word list when translating the application-specific vocabulary names of the abstract grammar.

The source code of the generated word list is illustrated in figure 3.

3.2.2 Generating the paradigm functions

In GF, a paradigm is a function that produces an inflection table. To generate the function for an extracted paradigm description, we need to specify its name, interface and body.

The name of the function is the paradigm's ID from the LMF, where the prefix `as` is simply replaced with `mk`.

The function's interface is declared in the `oper` section and parameter types in the `param` section. The names and values of the parameters reflect the attributes and values of the grammatical features in the lMF, with minor modifications. The `oper` definition is largely templatic, reflecting only the parts of speech from the LMF.

To help readers navigate the code, the paradigms are ordered by their part of speech and headers are generated in the form of comments for each part of speech section. These are the only comments generated at the moment.

Every paradigm is split into two separate functions a high-level dispatch function (`mkTšiutto`, and low-level to `mkTšiuttoConcrete`). This is to allow a developer to use the low-level function for debugging or testing purposes.

The high-level function takes a string with the chosen lemma form as its input and splits it into the technical stem parts, which are then simply delegated to the low-level function. An error message is thrown in case the input string is not able to match.

The names of the variables holding the technical stem parts are taken from the paradigm's name.

The low-level, or `concrete`, function is the one that generates the inflection table. On the left-hand side of the table is the grammatical features and on the right-hand side are the concatenation patterns that instantiates the wordforms.

Note that GF's implementation of regular expressions is non-greedy as opposed to many other programming languages. Because of this the expression `@(-(_+"t"+_))` is appended to the last part of the pattern. This expression is automatically created and makes the `` `t` `` match the last letter `t` in the wordform. This matching strategy is suspect to be language dependent.

The source code of the generated morphology module for our example is shown in figure 4.

It can be noticed that the language code is used for naming the module (`MorphoVot`).

3.3 Giellatekno infrastructure source code integration

The Giellatekno infrastructure has been characterized in Moshagen et al. (2013) to be a *development environment infrastructure* (as opposed to a resource infrastructure), offering a framework for building language-specific analysers and directly turn them into a wide range of useful programs.

From the point of view of our work on Votic morphology, the programs of interest are proofing tools and morphological analyzers.

For the integration with Giellatekno's infrastructure, several components are needed: a lexicon, paradigm descriptions in FST, automatic test declarations, and a Makefile that binds all the components together.

At the work's current stage all but the Makefile is being generated. Because of this, each component has only been tested on its own but not integrated in the infrastructure.

We don't include any examples of generated code for the Giellatekno infrastructure because of space considerations. We will only present and discuss the main design choices made.

3.3.1 Generating the lexicon

Instead of generating the lexicon straight into FST representation, we use Giellatekno's Sanat XML format. This is achieved by simple XML transformations from the LMF format.

The Giellatekno infrastructure uses the Sanat XML to generate its own source codes.

In its essence, the Sanat XML contains the lexical entry's lemma, the name of the paradigm (e.g a FST continuation class) and the technical stem needed by the continuation class.

What is crucially missing from our implementation is the Finnish translation equivalents used as interlingual pivots in the Giellatekno infrastructure. These equivalents will be generated when our work on the Votic morphological dictionary resource has reached the stage of adding them.

3.3.2 Generating the paradigm functions

The FST source code generated for the paradigm functions follow the general structure for realizational morphology introduced in (Karttunen, 2003). This differs from Giellatekno's tradition of using two-level rules as introduced in Koskenniemi (1983). This choice should not reflect any ideological stance, it was chosen only for pragmatic reasons.

A design decision was made so that currently only the low-level paradigm functions are generated. The segmenting of the lemma is done when generating the dictionary.

3.3.3 Generating tests

Tests are optional but used in Giellatekno's infrastructure for verifying that the paradigm functions work as intended.

For verifying generated wordforms they need to be matched with known to be correct wordforms. For this purpose the inflected wordforms stored in the original dictionary are used.

Note that the "correct" wordforms are currently the ones generated by the Morphology Lab. This means that the tests now only validate the performance of the generated FST paradigm functions. This could in the future be enhanced when information about corpus attestations or prescriptive information has been added to the Votic dictionary.

3.3.4 The Makefile

Giellatekno's template for the Makefile that orchestrates the building of all the components uses two-level of the Giellatekno infrastructure. It is currently not generated by the presented system and will not be discussed here.

4 Discussion

It is not yet known, which way of building language technology is the best or most beneficial for different stakeholders. The key points that will be discussed here, with regard to the presented work, is centered around the following four aspects: 1) work on morphology is done through an user interface similar to Wiktionary 2) work on a lexical resource is put in the middle 3) the general benefits of source code generation 4) inherent lemma form agnosticity.

The work on morphology is reduced to an interface similar to Wiktionary. In the presented approach, all that is needed by the user working on the morphological resource is knowledge of *wordforms*, not *morphology per se*.

This interface brings together the work on creating a normative morphology for Votic, but opens the process up for collaboration with the Votic language community.

Setting up a project in this way, the computational linguist could earlier start working on a higher level (e.g. describing the syntax) and delegate work on the morphology to language activists. By developing traditional GF applications, like the city or foods phrasebooks, the linguist could show the direct effect of the activists adding words to the dictionary and how the words show up in multiple phrases of the phrasebook. This direct connection is hoped to motivate the activists.

Other "motivators" are the benefits gained from integration with Giellatekno's infrastructure, mainly the proofing tools. Even though this is an indirect link, it is a parallel and simultaneous benefit.

Advantages of putting the resource in the middle. The key advantage of the presented approach springs out of putting the lexical resource in the middle. Here we keep in mind the LMF representation of the lexical resource. This centrality has several implications.

One aspect is the technological neutrality this brings. The resource models the procedural (or operational) knowledge of morphology, but this is not reflected in the shape of the resource. The resource does not use terminology from Finite State Transducers, nor from functional programming, even though the resource is used as an ontology for source code generation into both these programming paradigms.

Because of this neutrality, the lexical resource is believed to be readable and understandable in a 50 year perspective, whereas any program code implementing the same kind of knowledge might not be. In fact, a working platform for executing the code will most likely not even exist after 50 years.

The advantage of this self-documenting aspect is believed to be situated somewhere between the areas of computational linguistics, descriptive linguistics and documentational linguistics. Because of the close connection to language documentation, further research in this area is believed to be a worthwhile endeavor by the authors, especially for small and under-resourced languages.

Another advantage of putting the resource in the middle is that any faults found during the usage of the generated language technology, will get fixed up-streams in the lexical resource itself. In this way it is the resource that gets worked on, not only the language technology.

This is highly connected to the benefits of source code generation, discussed next.

General benefits of source code generation. The general benefits of using source code generation is that changes need only be done in one place. The usefulness of this gets larger for every added target platform, our work comprises only two.

By only needing to change faults in one central place might also encourage in the future to keep what is then considered "old" software functioning and up to date.

Not only fixing any found faults or simply improving the resource needs to be done in one central place. The centrality of code also ensures consistent style in the generated source code. Changing style throughout all the generated code is done by changing only in one place: in the source code generator.

Benefits of lemma-form agnosticity. The lemmaform agnosticity of the presented approach is seen beneficial to languages that lack a lexicographic tradition. Since Extract Morphology is independent of the choice of which wordform is used as lemma, this choice can be post-poned. This could permit more time and resources to investigate appropriate forms by linguists or lexicographers, or simply give more time to the real dictionary users to get acquainted using the dictionary.

Acknowledgments

Acknowledgments should be un-numbered last section. Do not include acknowledgements in anonymised review version.

References

Malin Ahlberg, Markus Forsberg, and Mans Hulden. 2014. Semi-supervised learning of morphological paradigms and lexicons. In *Proc. of EACL*. pages 569–578. http://www.aclweb.org/anthology/E/E14/E14-1.pdf#page=595.

Malin Ahlberg, Markus Forsberg, and Mans Hulden. 2015. Paradigm classification in supervised learning of morphology. In *HLT-NAACL*. pages 1024–1029. http://www.aclweb.org/old_anthology/N/N15/N15-1107.pdf.

August Ahlqvist. 1856. *Wotisk grammatik jemte språkprof och ordförteckning: (Föredr. d. 15 Oktober 1855).* s.n, [Helsingfors.

Paul Ariste. 1968. *A grammar of the Votic language.* Number vol. 68 in Indiana University publications. Uralic and Altaic series. Indiana University ; Mouton, Bloomington : The Hague.

Markus Forsberg. 2016. What can we learn from inflection tables? In *BAULT 2016*. Helsinki. http://blogs.helsinki.fi/language-technology/news/bault-2016/.

Markus Forsberg and Mans Hulden. 2016. Deriving Morphological Analyzers from Example Inflections. In *LREC*. http://www.lrec-conf.org/proceedings/lrec2016/pdf/1134_Paper.pdf.

Gil Francopoulo. 2013. *LMF lexical markup framework.* ISTE Ltd ; John Wiley & Sons, London; Hoboken, NJ.

Silja Grünberg, Ada Ambus, Georg Grünberg, Roman Kallas, Tatjana Murnikova, Tatjana Nikitina, Agnia-Agnes Reitsak, Savvati Smirnov, Indrek Hein, Esta Prangel, and Esta Prangel, editors. 2013. *Vadja keele sõnaraamat =: Vaďďaa tšeelee sõna-tširja = Словарь водского языка.* Eesti Keele Sihtasutus, Tallinn, 2., täiend. ja parand. tr edition.

Mans Hulden. 2014. Generalizing inflection tables into paradigms with finite state operations. In *Proceedings of the 2014 Joint Meeting of SIGMORPHON and SIGFSM*. pages 29–36. http://aclweb.org/anthology/W/W14/W14-2804.pdf.

Lauri Karttunen. 2003. Computing with realizational morphology. In *Computational linguistics and intelligent text processing*, Springer, pages 203–214.

Kimmo Koskenniemi. 1983. Two-Level Model for Morphological Analysis. In *IJCAI*. volume 83, pages 683–685. http://ijcai.org/Past%20Proceedings/IJCAI-83-VOL-2/PDF/020.pdf.

Sjur N. Moshagen, Tommi A. Pirinen, and Trond Trosterud. 2013. Building an open-source development infrastructure for language technology projects. *In: Proceedings of the 19th Nordic Conference of Computational Linguistics (NODALIDA 2013). Volume 16 of NEALT Proceedings Series. (May 22–24 2013)* pages 343–352.

Aarne Ranta. 2008. *Grammars as software libraries.*

Aarne Ranta. 2011. *Grammatical framework : programming with multilingual grammars.* Studies in computational linguistics. CSLI Publications.

Dmitri Tsvetkov. 1995. *Vatjan kielen Joenperän murteen sanasto.* Suomalais-ugrilainen seura ;Kotimaisten kielten tutkimuskeskus, Helsinki.

Dmitri Tsvetkov. 2008. *Vadja Keele Grammatika.* Eesti Keele Sihtasutus, Tallinn.

Елена Борисовна Маркус and Федор Иванович Рожанский. 2011. *Современный водский язык: тексты и грамматический очерк. Том 2, Грамматический очерк и библиография: [в 2-х томах].* Нестор-История, Санкт Петербург.

```
<LexicalEntry morphologicalPatterns="asTšiutto">
  <feat att="partOfSpeech" val="nn"/>
  <feat att="karp-lemgram" val="tšiutto..nn.1"/>
  <Lemma>
    <feat att="writtenForm" val="tšiutto"/>
  </Lemma>
  <WordForm>
    <feat att="writtenForm" val="tšiutto"/>
    <feat att="grammaticalNumber" val="singular"/>
    <feat att="grammaticalCase" val="nominative"/>
  </WordForm>
  <WordForm>
    <feat att="writtenForm" val="tšiutod"/>
    <feat att="grammaticalNumber" val="plural"/>
    <feat att="grammaticalCase" val="nominative"/>
  </WordForm>
</LexicalEntry>
```

Figure 1: LMF representation of the (toy example) lexical entry for *tšiutto*.

```
<MorphologicalPattern>
  <feat att="id" val="asTšiutto"/>
  <feat att="partOfSpeech" val="nn"/>
  <AttestedParadigmVariableSets>
    <AttestedParadigmVariableSet>
      <feat att="first-attest" val="tšiutto..nn.1"/>
      <feat att="1" val="tšiut"/>
      <feat att="2" val="o"/>
    </AttestedParadigmVariableSet>
  </AttestedParadigmVariableSets>
  <TransformSet>
    <GrammaticalFeatures>
      <feat att="grammaticalNumber" val="singular"/>
      <feat att="grammaticalCase" val="nominative"/>
    </GrammaticalFeatures>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddVariable"/>
      <feat att="variableNum" val="1"/>
    </Process>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddConstant"/>
      <feat att="stringValue" val="t"/>
    </Process>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddVariable"/>
      <feat att="variableNum" val="2"/>
    </Process>
  </TransformSet>
  <TransformSet>
    <GrammaticalFeatures>
      <feat att="grammaticalNumber" val="plural"/>
      <feat att="grammaticalCase" val="nominative"/>
    </GrammaticalFeatures>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddVariable"/>
      <feat att="variableNum" val="1"/>
    </Process>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddVariable"/>
      <feat att="variableNum" val="2"/>
    </Process>
    <Process>
      <feat att="operator" val="addAfter"/>
      <feat att="processType" val="pextractAddConstant"/>
      <feat att="stringValue" val="d"/>
    </Process>
  </TransformSet>
<MorphologicalPattern>
```

Figure 2: LMF representation of the (toy example) paradigm common for both *tšiutto* and *katto* (the concatenation patterns $x_1 \oplus \mathbf{t} \oplus x_2$ and $x_1 \oplus x_2 \oplus \mathbf{d}$).

```
fun
  lin tšiutto_N = mkTšiutto "tšiutto" ;
  lin katto_N = mkTšiutto "katto"
```

Figure 3: Generated source code for the Votic GF lexicon.

```
resource MorphoVot = {

param
  Number = singular | plural ;
  Case = nominative ;
  NForm = NF Number Case ;

oper
  Noun : Type = {s : NForm => Str} ;

-----------------------------------------------
-- Start of Noun section
-----------------------------------------------

  mkTšiutto : Str -> Noun = \tšiutto ->
    case tšiutto of {
      tšiut + "t" + o@(-(_+"t"+_)) => mkTšiuttoConcrete tšiut o ;
      _ => Predef.error "Unsuitable lemma for mkTšiutto"
    } ;

  mkTšiuttoConcrete : Str -> Str -> Noun = \tšiut,o ->
    { s =
      table {
        NF singular nominative => tšiut + "t" + o ;
        NF plural nominative => tšiut + o + "d"
      }
    } ;
}
```

Figure 4: Generated source code for the Votic GF morphology module.