MorphoDiTa: Morphological Dictionary and Tagger

Version 1.9.2

Contents

1 Introduction

MorphoDiTa: Morphological Dictionary and Tagger is an open-source tool for morphological analysis of natural language texts. It performs morphological analysis, morphological generation, tagging and tokenization and is distributed as a standalone tool or a library, along with trained linguistic models. In the Czech language, MorphoDiTa achieves state-of-the-art results with a throughput around 10-200K words per second. MorphoDiTa is a free software under Mozilla Public License 2.0 and the linguistic models are free for non-commercial use and distributed under CC BY-NC-SA license, although for some models the original data used to create the model may impose additional licensing conditions. MorphoDiTa is versioned using Semantic Versioning.

2 Online

2.1 Online Demo

Online demo is available as one of LINDAT/CLARIN services.

2.2 Web Service

Web service is also available as one of LINDAT/CLARIN services.

3 Release

3.1 Download

MorphoDiTa releases are available on GitHub, either as a pre-compiled binary package, or source code only. The binary package contains Linux, Windows and OS X binaries, Java bindings binary, C# bindings binary, and source code of MorphoDiTa and all language bindings). While the binary packages do not contain compiled Python or Perl bindings, packages for those languages are available in standard package repositories, i.e. on PyPI and CPAN.

- Latest release
- All releases, Changelog

3.1.1 Language Models

To use MorphoDiTa, a language model is needed. The language models are available from LINDAT/CLARIN infrastructure and described further in the MorphoDiTa User's Manual. Currently the following language models are available:

- Czech: czech-morfflex-pdt-160222 (documentation) czech-morfflex-pdt-131112 (documentation)
- English: english-morphium-wsj-140407 (documentation)

3.2 License

MorphoDiTa is an open-source project and is freely available for non-commercial purposes. The library is distributed under Mozilla Public License 2.0 and the associated models and data under CC BY-NC-SA, although for some models the original data used to create the model may impose additional licensing conditions.

If you use this tool for scientific work, please give credit to us by referencing MorphoDiTa website and Straková et al. 2014.

3.3 Platforms and Requirements

MorphoDiTa is available as a standalone tool and as a library for Linux/Windows/OS X. It does not require any additional libraries. As any supervised machine learning tool, it needs trained linguistic models to perform morphological analysis. The models for the Czech language are available with the tool.

4 MorphoDiTa Installation

MorphoDiTa releases are available on GitHub, either as a pre-compiled binary package, or source code only. The binary package contains Linux, Windows and OS X binaries, Java bindings binary, C# bindings binary, and source code of MorphoDiTa and all language bindings. While the binary packages do not contain compiled Python or Perl bindings, packages for those languages are available in standard package repositories, i.e. on PyPI and CPAN.

To use MorphoDiTa, a language model is needed. Here is a list of available language models.

If you want to compile MorphoDiTa manually, sources are available on on GitHub, both in the pre-compiled binary package releases and in the repository itself.

4.1 Requirements

- `G++ 4.7` or newer, `clang 3.2` or newer, Visual C++ 2015 or newer
- `make`
- `SWIG` or newer for language bindings other than `C++`

4.2 Compilation

To compile MorphoDiTa, run `make` in the `src` directory.

Make targets and options:
- `exe`: compile the binaries (default)
- `server`: compile the REST server
- `tools`: compile various tools
- `lib`: compile MorphoDiTa library (decoding only)
- `BITS=32` or `BITS=64`: compile for specified 32-bit or 64-bit architecture instead of the default one
- `MODE=release`: create release build which statically links the C++ runtime and uses LTO
- `MODE=debug`: create debug build
- `MODE=profile`: create profile build

4.2.1 Platforms

Platform can be selected using one of the following options:
- `PLATFORM=linux`, `PLATFORM=linux-gcc`: gcc compiler on Linux operating system, default on Linux
- `PLATFORM=linux-clang`: clang compiler on Linux, must be selected manually
- `PLATFORM=osx`, `PLATFORM=osx-clang`: clang compiler on OS X, default on OS X; `BITS=32+64` enables multiarch build
- `PLATFORM=win`, `PLATFORM=win-gcc`: gcc compiler on Windows (TDM-GCC is well tested), default on Windows
- `PLATFORM=win-vs`: Visual C++ 2015 compiler on Windows, must be selected manually; note that the `cl.exe` compiler must be already present in `PATH` and corresponding `BITS=32` or `BITS=64` must be specified

Either POSIX shell or Windows CMD can be used as shell, it is detected automatically.

4.2.2 Further Details

MorphoDiTa uses C++ BuilTem system, please refer to its manual if interested in all supported options.

4.3 Other language bindings

4.3.1 C#

Binary C# bindings are available in MorphoDiTa binary packages.

To compile C# bindings manually, run `make` in the `bindings/csharp` directory, optionally with the options descriged in MorphoDiTa Installation.

4.3.2 Java

Binary Java bindings are available in MorphoDiTa binary packages.

To compile Java bindings manually, run `make` in the `bindings/java` directory, optionally with the options descriged in MorphoDiTa Installation. Java 6 and newer is supported.

The Java installation specified in the environment variable `JAVA_HOME` is used. If the environment variable does not exist, the `JAVA_HOME` can be specified using

```
make JAVA_HOME=path_to_Java_installation
```

4.3.3 Perl

The Perl bindings are available as `Ufal-MorphoDiTa` package on CPAN.

To compile Perl bindings manually, run `make` in the `bindings/perl` directory, optionally with the options descriged in MorphoDiTa Installation. Perl 5.10 and later is supported.

Path to the include headers of the required Perl version must be specified in the `PERL_INCLUDE` variable using

```
make PERL_INCLUDE=path_to_Perl_includes
```

4.3.4 Python

The Python bindings are available as `ufal.morphodita` package on PyPI.

To compile Python bindings manually, run `make` in the `bindings/python` directory, optionally with options descriged in MorphoDiTa Installation. Both Python 2.6+ and Python 3+ are supported.

Path to the include headers of the required Python version must be specified in the `PYTHON_INCLUDE` variable using

```
make PYTHON_INCLUDE=path_to_Python_includes
```

5 MorphoDiTa User's Manual

In a natural language text, the task of morphological analysis is to assign for each token (word) in a sentence its lemma (cannonical form) and a part-of-speech tag (POS tag). This is usually achieved in two steps: a morphological dictionary looks up all possible lemmas and POS tags for each word, and subsequently, a morphological tagger picks for each word the best lemma-POS tag candidate. The second task is called a disambiguation.

MorphoDiTa also performs these two steps of morphological analysis: It first outputs all possible pairs of lemma and POS tag for each token. Consequently, the optimal combination of lemmas and POS tags is selected for the words in a sentence using an algorithm described in Spoustová et al. 2009.

Like any supervised machine learning tool, MorphoDiTa needs a trained linguistic model. This section describes the available language models and also the commandline tools and interfaces. The C++ library is described elsewhere, either in MorphoDiTa API Tutorial or in MorphoDiTa API Reference.

5.1 Czech MorphoDiTa Models

Czech models are distributed under the CC BY-NC-SA licence. The Czech morphology uses the MorfFlex CZ 160310 Czech morphological dictionary and the Czech tagger is trained on PDT 3.0. Czech models work in MorphoDiTa version 1.0 or later.

Apart from MorfFlex CZ dictionary, a prefix guesser and statistical guesser are implemented and can be optionally used when performing morphological analysis. The version 160310 cotains also models with DeriNet as a morphological derivator (requiring MorphoDiTa 1.9 or later).

Czech models are versioned according to the version of the MorfFlex CZ morphological dictionary used, the version format is `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 160310.

Compared to Featurama http://sourceforge.net/projects/featurama/ (state-of-the-art Czech tagger implementation), the models are 5 times faster and 10 times smaller.

5.1.1 Download

The latest version 160310 of the Czech MorphoDiTa models can be downloaded from LINDAT/CLARIN repository.

Previous Versions

- Version 131112 of the Czech MorphoDiTa models can be downloaded from LINDAT/CLARIN repository.

5.1.2 Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project *LM2010013*).

The Czech morphological system was devised by Jan Hajič.

The MorfFlex CZ dictionary was created by Jan Hajič and Jaroslava Hlaváčová.

The morphological guesser research was supported by the projects *1ET101120503* and *1ET101120413* of Academy of Sciences of the Czech Republic and *100008/2008* of Charles University Grant Agency. The research was performed by Jan Hajič, Jaroslava Hlaváčová and David Kolovratník.

The tagger algorithm and feature set research was supported by the projects *MSM0021620838* and *LC536* of Ministry of Education, Youth and Sports of the Czech Republic, *GA405/09/0278* of the Grant Agency of the Czech Republic and *1ET101120503* of Academy of Sciences of the Czech Republic. The research was performed by Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab and Miroslav Spousta.

The tagger is trained on morphological layer of Prague Dependency Treebank PDT 3.0, which was supported by the projects *LM2010013*, *LC536*, *LN00A063* and *MSM0021620838* of Ministry of Education, Youth and Sports of the Czech Republic, and developed by Martin Buben, Jan Hajič, Jiří Hana, Hana Hanová, Barbora Hladká, Emil Jeřábek, Lenka Kebortová, Kristýna Kupková, Pavel Květoň, Jiří Mírovský, Andrea Pfimpfrová, Jan Štěpánek and Daniel Zeman.

The morphological derivator is based on DeriNet, which was supported by the Grant No. 16-18177S of the Grant Agency of the Czech Republic and uses language resources developed, stored, and distributed by the LINDAT/CLARIN project of the Ministry of Education, Youth and Sports of the Czech Republic (project LM2015071).

Publications

- (Hajič 2004) Jan Hajič. *Disambiguation of Rich Inflection: Computational Morphology of Czech.* Karolinum Press (2004).
- Hlaváčová Jaroslava, Kolovratník David. *Morfologie češtiny znovu a lépe.* In Informačné Technológie - Aplikácie a Teória. Zborník príspevkov, ITAT 2008. Seňa, Slovakia: PONT s.r.o., 2008, pp. 43-47.
- (Spoustová et al. 2009) Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, Miroslav Spousta. 2009. *Semi-Supervised Training for the Averaged Perceptron POS Tagger.* In Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pages 763-771, Athens, Greece, March. Association for Computational Linguistics.
- (Straková et al. 2014) Straková Jana, Straka Milan and Hajič Jan. *Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition.* In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 13-18, Baltimore, Maryland, June 2014. Association for Computational Linguistics.
- (Žabokrtský et al. 2016) Zdeněk Žabokrtský, Magda Ševčíková, Milan Straka, Jonáš Vidra and Adéla Limburská. *Merging Data Resources for Inflectional and Derivational Morphology in Czech.* In Proceedings of the Tenth International Conference on Language Resources and Evaluation (LREC 2016), Portorož, Slovenia, May 2016.

5.1.3 Czech Morphological System

In the Czech language, MorphoDiTa uses Czech morphological system by Jan Hajič (Hajič 2004). In this system, which we call *PDT tag set*, the tags are positional with 15 positions corresponding to part of speech, detailed part of speech, gender, number, case, etc. (e.g. `NNFS1-----A----`). Different meanings of same lemmas are distinguished and additional comments can be provided for every lemma meaning. The lemma itself without the comments and meaning specification is called a *raw lemma*. The following examples illustrate this:

- `Japonsko_;G` (raw lemma: `Japonsko`)
- `se_^(zvr._zájmeno/částice)` (raw lemma: `se`)
- `tvořit_:T` (raw lemma: `tvořit`)

For a more detailed reference about the Czech morphology, please see Lemma and Tag Structure in PDT 2.0.

5.1.4 Main Czech Model

The main Czech model contains the following files:

`czech-morfflex-160310.dict`
: Morphological dictionary based on the Jan Hajič's (Hajič 2004) system with PDT tag set created from MorfFlex CZ 160310 morphological dictionary.

`czech-morfflex-pdt-160310.tagger`
: Tagger trained on the training portion of PDT 3.0 using the `neopren` feature set. It contains the

czech-morfflex-160310.dict morphological dictionary. and reaches 95.57% tag accuracy, 97.75% lemma accuracy and 94.93% overall accuracy on PDT 3.0 etest data (whose morphological tags and lemmas were remapped using the czech-morfflex-160310.dict dictionary). Model speed: ~10k words/s, model size: 17MB.

5.1.5 Part of Speech Only Variant

The PDT tag set used by the main Czech model is very fine-grained. In many situations, only the part of speech tags would be sufficient. Therefore, we provide a variant of the model, denoted as `pos_only`, where only the first two characters of the fifteen-letter tags are used, representing the part of speech and detailed part of speech, respectively. There are 67 such two-letter tags.

`czech-morfflex-160310-pos_only.dict`
: Morphological dictionary based on the Jan Hajič's (Hajič 2004) system created from MorfFlex CZ 160310 morphological dictionary. Only first two tag characters of PDT tag set are used.

`czech-morfflex-pdt-160310-pos_only.tagger`
: Very fast tagger trained on the training portion of PDT 3.0 using the `neopren` feature set. It contains the `czech-morfflex-160310-pos_only.dict` morphological dictionary and reaches 99.04% tag accuracy, 97.62% lemma accuracy and 97.56% overall accuracy on PDT 3.0 etest data (which morphological tags and lemmas were remapped using the `czech-morfflex-160310-pos_only.dict` dictionary). Model speed: ~200k words/s, model size: 4MB.

5.1.6 No Diacritical Marks Variant

Sometimes the text to be analyzed does not contain diacritical marks. We therefore provide variants of the morphological dictionary and tagger for this purpose – morphological analysis, morphological generation and tagging employ forms without diacritical marks. Note that the lemmas do have diacritical marks.

We provide the `no_dia` variants for all four models described above:

`czech-morfflex-160310-no_dia.dict`
: No diacritical marks variant of `czech-morfflex-160310.dict`.

`czech-morfflex-pdt-160310-no_dia.tagger`
: No diacritical marks variant of `czech-morfflex-160310.tagger`. It reaches 94.74% tag accuracy, 97.05% lemma accuracy and 93.83% overall accuracy on PDT 3.0 etest data (which morphological tags and lemmas were remapped using the `czech-morfflex-160310-no_dia.dict` dictionary) with diacritical marks removed. Model speed: ~5k words/s, model size: 21MB.

`czech-morfflex-160310-no_dia-pos_only.dict`
: No diacritical marks variant of `czech-morfflex-160310-pos_only.dict`.

`czech-morfflex-pdt-160310-no_dia-pos_only.tagger`
: No diacritical marks variant of `czech-morfflex-160310-pos_only.tagger`. It reaches 98.59% tag accuracy, 97.04% lemma accuracy and 96.96% overall accuracy on PDT 3.0 etest data (which morphological tags and lemmas were remapped using the `czech-morfflex-160310-no_dia-pos_only.dict` dictionary) with diacritical marks removed. Model speed: ~130k words/s, model size: 9MB.

5.1.7 Morphological Derivation Model using DeriNet

All version 160310 models are available also with morphological derivator using DeriNet version 1.1. The models which include DeriNet require MorphoDiTa 1.9 or later.

In the future, only the models with DeriNet will be released.

5.1.8 Models with Raw Lemmas

The Czech morphological system distinguish different meanings of same lemmas by numbering the lemmas with multiple meanings and supplying additional comments for every lemma meaning, as described and demonstrated in Czech Morphological System. Sometimes this may be undesirable, for example when comparing to systems which do not use the MorfFlex CZ 160310 morphological dictionary.

To obtain lemmas without any additional information (*raw lemmas* in terms of MorphoDiTa API), use `strip_lemma_id` tag set converter. Previously, specific dictionary and tagger model variants were provided, which is not needed anymore.

5.1.9 Czech Model History

`czech-morfflex-160310` and `czech-morfflex-pdt-160310` (require MorphoDiTa 1.0 or later)
Trained on PDT 3.0 using MorfFlex CZ 160310, variants: Part of Speech Only, No Diacritical Marks. Download from LINDAT/CLARIN repository.

`czech-morfflex-131112` and `czech-morfflex-pdt-131112` (require MorphoDiTa 1.0 or later)
Trained on PDT 2.5 using MorfFlex CZ 131112, variants Part of Speech Only, Raw Lemmas. Download from LINDAT/CLARIN repository.

5.2 English MorphoDiTa Models

English models are created using the following data:

- *SCOWL (Spell Checker Oriented Word Lists)*: This word list is used in morphological generation to create all possible word forms of a given word.

  Copyright: *Copyright 2000-2011 by Kevin Atkinson. Permission to use, copy, modify, distribute and sell these word lists, the associated scripts, the output created from the scripts, and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Kevin Atkinson makes no representations about the suitability of this array for any purpose. It is provided "as is" without express or implied warranty.*

- *Wall Street Journal*, part of the Penn Treebank 3: Morphologically annotated texts which are commonly used to train English POS tagger.

  Licensing: Available as LDC99T42 in LDC catalog under LDC User Agreement.

The resulting models are distributed under the CC BY-NC-SA licence. English models work in MorphoDiTa version 1.1 or later.

English models are versioned according to the release date, the version format is `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 140407.

5.2.1 Download

The latest version 140407 of the English MorphoDiTa models can be downloaded from LINDAT/CLARIN repository.

5.2.2 Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project *LM2010013*).

The morphological POS analyzer development was supported by grant of the Ministry of Education, Youth and Sports of the Czech Republic No. *LC536* "Center for Computational Linguistics".

The morphological POS analyzer research was performed by Johanka Spoustová (Spoustová 2008; the `Treex::Tool::EnglishMorpho::Analysis` Perl module). The lemmatizer was implemented by Martin Popel (Popel 2009; the `Treex::Tool::EnglishMorpho::Lemmatizer` Perl module). The lemmatizer is based on `morpha`, which was released under LGPL licence as a part of RASP system.

The tagger algorithm and feature set research was supported by the projects *MSM0021620838* and *LC536* of Ministry of Education, Youth and Sports of the Czech Republic, *GA405/09/0278* of the Grant Agency of the Czech Republic and *1ET101120503* of Academy of Sciences of the Czech Republic. The research was performed by Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab and Miroslav Spousta.

Publications

- (Popel 2009) Martin Popel. *Ways to Improve the Quality of English-Czech Machine Translation.* Master Thesis at Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague (2009).
- (Spoustová 2008) Drahomíra "johanka" Spoustová. *Morphium – morphological analyser for Penn treebank POS tagset.* Perl Software developed at Institute of Formal and Applied Linguistics, Faculty of Mathematics and Physics, Charles University in Prague (2008).
- (Spoustová et al. 2009) Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, Miroslav Spousta. 2009. *Semi-Supervised Training for the Averaged Perceptron POS Tagger.* In Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pages 763-771, Athens, Greece, March. Association for Computational Linguistics.
- (Straková et al. 2014) Straková Jana, Straka Milan and Hajič Jan. *Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition.* In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 13-18, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

5.2.3 English Morphological System

The English morphology uses standard Penn Treebank POS tags. Nevertheless, the lemma structure is unique:

- The lemmatizer recognizes negative prefixes and removes it from the lemma. In terms of MorphoDiTa API, *raw lemma* is the lemma without negative prefix.
- The negative prefix is also stored to allow morphological generation of word form with the same negative prefix. In terms of MorphoDiTa API, *lemma id* is the raw lemma plus the negative prefix.

The negative prefix is separated from the (always nonempty) lemma using a `^` character (`able^un`). During morphological generation, the negative prefix is honored. Furthermore, when the lemma ends with `^` (i.e., negative prefix is empty, as in `able^`), forms with negative prefixes are generated. It is also possible to generate all forms without any negative prefix by appending `+` after the lemma (for example `able+`).

5.2.4 English Model

The English model contains the following files:

`english-morphium-<version>.dict`
: Morphological dictionary. The SCOWL word list has been automatically analyzed and lemmatized and uses as the dictionary. The guesser performing the analyzation and lemmatization is available.

`english-morphium-wsj-<version>.tagger`
: Tagger trained on the training portion of Wall Street Journal (Sections 0-18) and tuned on the development portion (Sections 19-21). Contains the `english-morphium-<version>.dict` morphological dictionary.

  The latest version `english-morphium-wsj-140407.tagger` reaches 97.27% tag accuracy on Wall Street Journal test portion (Section 22-24). Model speed: ˜60k words/s, model size: 6MB.

5.2.5 No Negations Variant

Stripping of negative prefixes (or handling the lemmas with negative prefixes stripped) may not be desirable. Therefore, a variant of the English model denoted by `no_negation` is provided, which does not strip negative prefixes from lemmas.

`english-morphium-<version>-no_negation.dict`
: Morphological dictionary which does not strip negative lemma prefixes. The SCOWL word list has been automatically analyzed and lemmatized and uses as the dictionary. The guesser performing the analyzation and lemmatization is available.

`english-morphium-wsj-<version>-no_negation.tagger`
: Tagger which does not strip negative lemma prefixes, trained on the training portion of Wall Street Journal (Sections 0-18) and tuned on the development portion (Sections 19-21). Contains the `english-morphium-<version>-no_negation.dict` morphological dictionary.

  The latest version `english-morphium-wsj-140407-no_negation.tagger` reaches 97.25% tag accuracy on Wall Street Journal test portion (Section 22-24). Model speed: ˜60k words/s, model size: 6MB.

5.2.6 English Model Changes

`english-morphium-140407` and `english-morphium-wsj-140407` (require MorphoDiTa 1.1 or later)
: Recognize also ”non-” as a negative prefix. Formerly, only ”non” was recognized.

`english-morphium-140304` and `english-morphium-wsj-140304` (require MorphoDiTa 1.0 or later)
: Initial release.

5.3 Running the Tagger

Probably the most common usage of MorphoDita is running a tagger to tag your data using
`run_tagger tagger_model`

The input is assumed to be in UTF-8 encoding and can be either already tokenized and segmented, or it can be a plain text which is tokenized and segmented automatically.

Any number of files can be specified after the `tagger_model`. If an argument `input_file:output_file` is used, the given `input_file` is processed and the result is saved to `output_file`. If only `input_file` is used, the result is saved to standard output. If no argument is given, input is read from standard input and written to standard output.

The full command syntax of `run_tagger` is

```
run_tagger [options] tagger_file [file[:output_file]]...
Options: --input=untokenized|vertical
         --convert_tagset=pdt_to_conll2009|strip_lemma_comment|strip_lemma_id
         --derivation=none|root|path|tree
         --guesser=0|1 (should morphological guesser be used)
         --output=vertical|xml
```

5.3.1 Input Formats

The input format is specified using the `--input` option. Currently supported input formats are:
- `untokenized` (default): the input is tokenized and segmented using a tokenizer defined by the model,
- `vertical`: the input is in vertical format, every line is considered a word, with empty line denoting end of sentence.

5.3.2 Tag Set Conversion

Some tag sets can be converted to different ones. Currently supported tag set conversions are:

- `pdt_to_conll2009`: convert Czech PDT tag set to CoNLL 2009 tag set,
- `strip_lemma_comment`: strip lemma comment (see Lemma Structure in API Reference),
- `strip_lemma_id`: strip lemma id (see Lemma Structure in API Reference).

5.3.3 Morphological Derivation

If the morphological model includes a morphological derivator, some morphological derivation operation may be performed on lemmas:

- `none` (default): no morphological derivation is performed
- `root`: lemma is replaced by its root in the morphological derivation tree
- `path`: lemma is replaced by a space separated path to its root in the morphological derivation tree (the original lemma is first, followed by its parent, with the root being the last one)
- `tree`: whole morphological derivation tree is appended after the lemma, encoded in the following way: root node is the first, then the subtrees of the root children are encoded recursively (each after one space), followed by a final space (which denotes that the children are complete)

5.3.4 Morphological Guesser

By default, every tagger model uses the morphological guesser settings employed during the model training. However, the usage of morphological guesser can be overridden by the `guesser` parameter.

5.3.5 Output Formats

The output format is specified using the `--output` option. Currently supported output formats are:

- `xml` (default): Simple XML format without a root element, using `<sentence>` element to mark sentences and `<token lemma="..." tag="...">...</token>` element to encode token and its assigned lemma and tag.

  Example output for input `Děti pojedou k babičce.  Už se těší.` (line breaks added):

```
<sentence><token lemma='dítě' tag='NNFP1-----A----'>Děti</token>
<token lemma='jet-1_^(pohybovat_se,_ne_však_chůzí)'
    tag='VB-P---3F-AA---'>pojedou</token>
<token lemma='k-1' tag='RR--3----------'>k</token>
<token lemma='babička' tag='NNFS3-----A----'>babičce</token>
<token lemma='.' tag='Z:-------------'>.</token></sentence>
<sentence><token lemma='už-1' tag='Db-------------'>Už</token>
<token lemma='se_^(zvr._zájmeno/částice)' tag='P7-X4----------'>se</token>
<token lemma='těšit_:T' tag='VB-S---3P-AA---'>těší</token>
<token lemma='.' tag='Z:-------------'>.</token></sentence>
```

- `vertical`: Every output line is a tag separated triple form-lemma-tag, with empty line denoting end of sentence.

  Example output for input `Děti pojedou k babičce.  Už se těší.`:

```
Děti    dítě    NNFP1-----A----
pojedou jet-1_^(pohybovat_se,_ne_však_chůzí)    VB-P---3F-AA---
k       k-1     RR--3----------
babičce babička NNFS3-----A----
.       .       Z:-------------

Už      už-1    Db-------------
se      se_^(zvr._zájmeno/částice)      P7-X4----------
těší    těšit_:T        VB-S---3P-AA---
.       .       Z:-------------
```

5.4 Running the Morphology

There are multiple commands performing morphological tasks. The `run_morpho_analyze` executable performs morphological analysis and the `run_morpho_generate` executable performs morphological generation. The output of these commands is suitable for automatic processing.

The `run_morpho_cli` executable performs both morphological analysis and generation, but is designed to be used interactively and produces more human-readable output.

5.4.1 Morphological Analysis

The morphological analysis can be performed by running

```
run_morpho_analyze morphology_model use_guesser
```

The input is assumed to be in UTF-8 encoding and can be either already tokenized and segmented, or it can be a plain text which is tokenized and segmented automatically. The input files are specified same as with the `run_tagger` command.

Some morphological models contain both a manually created dictionary and a guesser. Therefore, a numeric `use_guesser` argument is required. If non-zero, the guesser is used, otherwise not.

Because tagger models contain an embedded morphological model, a tagger model can be used instead of morphological one if `--from_tagger` option is specified.

The full command syntax of `run_morpho_analyze` is

```
run_morpho_analyze [options] morphology_model use_guesser [file[:output_file]]...
Options: --input=untokenized|vertical
         --convert_tagset=pdt_to_conll2009|strip_lemma_comment|strip_lemma_id
         --derivation=none|root|path|tree
         --output=vertical|xml
         --from_tagger
```

Input Formats

The input format is specified using the `--input` option. Currently supported input formats are:

- `untokenized` (default): the input is tokenized and segmented using a tokenizer defined by the model,
- `vertical`: the input is in vertical format, every line is considered a word, with empty line denoting end of sentence.

Note that the input data is also segmented, even if it is not strictly necessary. Therefore, the input is processed by whole paragraphs (ending by an empty line).

Tag Set Conversion

Some tag sets can be converted to different ones. Currently supported tag set conversions are:

- `pdt_to_conll2009`: convert Czech PDT tag set to CoNLL 2009 tag set,
- `strip_lemma_comment`: strip lemma comment (see Lemma Structure in API Reference),
- `strip_lemma_id`: strip lemma id (see Lemma Structure in API Reference).

Morphological Derivation

If the morphological model includes a morphological derivator, some morphological derivation operation may be performed on lemmas:

- `none` (default): no morphological derivation is performed

- `root`: lemma is replaced by its root in the morphological derivation tree
- `path`: lemma is replaced by a space separated path to its root in the morphological derivation tree (the original lemma is first, followed by its parent, with the root being the last one)
- `tree`: whole morphological derivation tree is appended after the lemma, encoded in the following way: root node is the first, then the subtrees of the root children are encoded recursively (each after one space), followed by a final space (which denotes that the children are complete)

Output Formats

The output format is specified using the `--output` option. Currently supported output formats are:

- `xml` (default): Simple XML format without a root element, using using `<token><analysis lemma="..." tag="..."/><analysis...>...</token>` element to encode morphological analysis.

  Example output for input `Děti pojedou k babičce. Už se těší.` (line breaks added):

```
<sentence><token><analysis lemma="dítě" tag="NNFP1-----A----"/><analysis lemma="dítě"
    tag="NNFP4-----A----"/><analysis lemma="dítě" tag="NNFP5-----A----"/>Děti</token>
<token><analysis lemma="jet-1_^(pohybovat_se,_ne_však_chůzí)"
    tag="VB-P---3F-AA---"/>pojedou</token>
<token><analysis lemma="k-1" tag="RR--3----------"/><analysis
    lemma="k-3_^(označení_pomocí_písmene)" tag="NNNXX-----A----"/><analysis
    lemma="k-4`kůň_:B_^(jednotka_výkonu)" tag="NNMXX-----A---8"/><analysis
    lemma="k-8_:B_^(ost._zkratka)" tag="XX------------8"/><analysis
    lemma="komanditní_:B_^(jen_komanditní_společnost)"
    tag="AAXXX----1A---8"/><analysis lemma="koncernový_:B"
    tag="AAXXX----1A---8"/><analysis lemma="kuo-1_:B_,t_^(stará_jednotka_výkonu)"
    tag="NNNXX-----A---8"/>k</token>
<token><analysis lemma="babička" tag="NNFS3-----A----"/><analysis lemma="babička"
    tag="NNFS6-----A----"/>babičce</token>
<token><analysis lemma="." tag="Z:-------------"/>.</token></sentence>
<sentence><token><analysis lemma="už-1" tag="Db-------------"/><analysis lemma="už-2"
    tag="TT-------------"/>Už</token>
<token><analysis lemma="se_^(zvr._zájmeno/částice)" tag="P7-X4----------"/><analysis
    lemma="s-1" tag="RV--2----------"/><analysis lemma="s-1"
    tag="RV--7----------"/>se</token>
<token><analysis lemma="těšit_:T" tag="VB-P---3P-AA---"/><analysis lemma="těšit_:T"
    tag="VB-S---3P-AA---"/>těší</token>
<token><analysis lemma="." tag="Z:-------------"/>.</token></sentence>
```

- `vertical`: Every output line contains a word and a tab separated lemma-tag pairs assigned to the input word, with empty line denoting end of sentence.

  Example output for input `Děti pojedou k babičce. Už se těší.`:

```
Děti    dítě    NNFP1-----A---- dítě    NNFP4-----A---- dítě    NNFP5-----A----
pojedou jet-1_^(pohybovat_se,_ne_však_chůzí)    VB-P---3F-AA---
k       k-1     RR--3---------- k-3_^(označení_pomocí_písmene)  NNNXX-----A----
    k-4`kůň_:B_^(jednotka_výkonu)   NNMXX-----A---8 k-8_:B_^(ost._zkratka)
    XX------------8 komanditní_:B_^(jen_komanditní_společnost)      AAXXX----1A---8
    koncernový_:B   AAXXX----1A---8 kuo-1_:B_,t_^(stará_jednotka_výkonu)
    NNNXX-----A---8
babičce babička NNFS3-----A---- babička NNFS6-----A----
.       .       Z:-------------

Už      už-1    Db------------- už-2    TT-------------
se      se_^(zvr._zájmeno/částice)      P7-X4---------- s-1     RV--2----------
    s-1     RV--7----------
těší    těšit_:T        VB-P---3P-AA--- těšit_:T        VB-S---3P-AA---
.       .       Z:-------------
```

5.4.2 Morphological Generation

The morphological generation can be performed by running
`run_morpho_generate morphology_model use_guesser`

The input is assumed to be in UTF-8 encoding. The input files are specified same as with the run_tagger command.

Input for morphological generation has to be in vertical format, each line containing a lemma, which can be optionally followed by a tab and a tag wildcard. The output has the same number of lines as input, line *l* contains tab separated form-lemma-tag triplets which can be generated from the lemma on he input line *l*. If a tag wildcard was provided, only triplets with matching tags are returned.

Some morphological models contain both a manually created dictionary and a guesser. Therefore, a numeric `use_guesser` argument is required. If non-zero, the guesser is used, otherwise not.

Because tagger models contain an embedded morphological model, a tagger model can be used instead of morphological one if `--from_tagger` option is specified.

The full command syntax of `run_morpho_generate` is

```
run_morpho_generate [options] morphology_model use_guesser [input_file[:output_file]]...
Options: --convert_tagset=pdt_to_conll2009|strip_lemma_comment|strip_lemma_id
         --from_tagger
```

Example input data:

```
dítě
jet     ?[fN]??[-1]
k-1
babička NNFS3-----A----
```

Example output:

```
dítě    dítě    NNNS1-----A---- dítě    dítě    NNNS4-----A---- dítě    dítě
    NNNS5-----A---- dítěte  dítě    NNNS2-----A---- dítěti  dítě    NNNS3-----A---- dítěti
    dítě    NNNS6-----A---- dítětem dítě    NNNS7-----A---- děti    dítě    NNFP1-----A----
    děti    dítě    NNFP4-----A---- děti    dítě    NNFP5-----A---- dětma   dítě
    NNFP7-----A---6 dětmi   dítě    NNFP7-----A---- dětem   dítě    NNFP3-----A---- dětí
    dítě    NNFP2-----A---- dětech  dítě    NNFP6-----A---- dětima  dítě_,h NNFP7-----A---6
ject    jet     Vf--------A---6 jet     jet-1_^(pohybovat_se,_ne_však_chůzí)
    Vf--------A---- jeti    jet-1_^(pohybovat_se,_ne_však_chůzí)    Vf--------A---2 nejet
    jet-1_^(pohybovat_se,_ne_však_chůzí)    Vf--------N---- nejeti
    jet-1_^(pohybovat_se,_ne_však_chůzí)    Vf--------N---2 jet
    jet-2_,h_^(letadlo_s_tryskovým_pohonem)NNIS1-----A----  jety
    jet-2_,h_^(letadlo_s_tryskovým_pohonem) NNIP1-----A----
k       k-1     RR--3---------- ke      k-1     RV--3---------- ku      k-1
    RV--3---------1
babičce babička NNFS3-----A----
```

Tag Set Conversion

Some tag sets can be converted to different ones. Currently supported tag set conversions are:

- `pdt_to_conll2009`: convert Czech PDT tag set to CoNLL 2009 tag set,
- `strip_lemma_comment`: strip lemma comment (see Lemma Structure in API Reference),
- `strip_lemma_id`: strip lemma id (see Lemma Structure in API Reference).

Note that the tag set conversion is applied only to the output, not to the input lemmas and wildcards.

Tag Wildcards

When only forms with a specific tag should be generated for a given lemma, tag wildcard can be specified. The tag wildcard is a simple wildcard allowing to filter the results of morphological generation.

Most characters of a tag wildcard match corresponding characters of a tag, with the following exceptions:

- `?` matches any character of a tag.
- `[chars]` matches any of the characters listed. The dash `-` has no special meaning and if `]` is the first character in `chars`, it is considered as one of the characters and does not end the group.
- `[^chars]` matches any of the characters *not* listed.

5.4.3 Interactive Morphological Analysis and Generation

Morphological analysis and generation which is interactive and more human readable can be run using:

```
run_morpho_cli morphology_model
```

The input is read from standard input, command on each line. If there is no tab on a line, analysis is performed on the given word. If there is a tab on a line, generation is performed on the first word, using the second word as a tag wildcard. If the second word is empty (i.e., the input is for example "on "), all forms are generated.

Because tagger models contain an embedded morphological model, a tagger model can be used instead of morphological one if `--from_tagger` option is specified.

The full command syntax of `run_morpho_cli` is

```
run_morpho_cli [options] morphology_model
Options: --from_tagger
```

5.5 Running the Tokenizer

Using the `run_tokenizer` executable it is possible to perform only tokenization and segmentation.

The input is a UTF-8 encoded plain text and the input files are specified same as with the `run_tagger` command.

The tokenizer can be specified either by using a morphology model (`--morphology` option), tagger model (`--tagger` option) or by using a tokenizer identifier (`--tokenizer` option). Currently supported tokenizer identifiers are:

- `czech`
- `english`
- `generic`

The full command syntax of `run_tokenizer` is

```
run_tokenizer [options] [file[:output_file]]...
Options: --tokenizer=czech|english|generic
         --morphology=morphology_model_file
         --tagger=tagger_model_file
         --output=vertical|xml
```

5.5.1 Output Formats

The output format is specified using the `--output` option. Currently supported output formats are:

- `xml` (default): Simple XML format without a root element, using <`sentence`> element to mark sentences and <`token`> element to mark tokens.

  Example output for input `Děti pojedou k babičce.  Už se těší.` (line breaks added):

  ```
  <sentence><token>Děti</token> <token>pojedou</token> <token>k</token>
  <token>babičce</token><token>.</token></sentence> <sentence><token>Už</token>
  <token>se</token> <token>těší</token><token>.</token></sentence>
  ```

- `vertical`: Each token is on a separate line, every sentence is ended by a blank line.

  Example output for input `Děti pojedou k babičce.  Už se těší.`:

  ```
  Děti
  pojedou
  k
  babičce
  .

  Už
  se
  těší
  .
  ```

5.6 Running REST Server

MorphoDiTa also provides REST server binary `morphodita_server`. The binary uses MicroRestD as a REST server implementation and provides MorphoDiTa REST API.

The full command syntax of `morphodita_server` is

```
morphodita_server [options] port (model_name weblicht_id model_file acknowledgements)*
Options: --daemon
```

The `morphodita_server` can run either in foreground or in background (when `--daemon` is used). The specified model files are loaded during start and kept in memory all the time. This behaviour might change in future to load the models on demand.

5.7 Custom Morphological and Tagging Models

It is possible to create custom morphological and tagging models.

5.7.1 Custom Morphological Models

Custom morphological models can be created using `encode_dictionary` binary.

The `encode_dictionary` reads from standard input and prints MorphoDiTa morphological model on standard output. The input of `encode_dictionary` is a textual representation of morphological dictionary. It should be UTF-8 encoded and every line should be a tab separated triplet `lemma \t tag \t form`. All forms of one lemma must appear in a continuous region and no line should appear more than once (`sort -u` can be used to achieve this).

Run `encode_dictionary` with the following options:

```
encode_dictionary generic max_suffix_len unknown_tag number_tag punctuation_tag symbol_tag
```

- `generic`: This parameter defines tokenizer and other language specific behaviour. Other values than `generic` take different options and are not documented.
- `max_suffix_len`: Maximum length of suffixes in automatically inferred inflexion classes. If unsure, use 8 (we use 8 for Czech and 4 for English). Smaller values produce larger and slightly faster models.
- `unknown_tag`: Assigned to a form during analysis if no matching tag can be found.
- `number_tag`: Assigned to a form during analysis if the form was not found in the dictionary and it looks like a number. Can be the same as `unknown_tag`.
- `punctuation_tag`: Assigned to a form during analysis if the form was not found in the dictionary and it consists of Unicode characters in the Punctuation category. Can be the same as `unknown_tag`.

- `symbol_tag`: Assigned to a form during analysis if the form was not found in the dictionary and it consists of Unicode characters in the Symbol category. Can be the same as `unknown_tag`.

Example input data:

```
dog     NN      dog
dog     NNS     dogs
go      VB      go
go      VBP     go
go      VBZ     goes
go      VBG     going
go      VBD     went
```

Example command line:

```
encode_dictionary generic 8 UNK NUM PUNC SYM <input_data >output_model
```

Using External Morphology

Sometimes it is useful to train MorphoDiTa tagger using external morphological analysis, without having a MorphoDiTa morphological dictionary.

That is possible using a so called *external morphology model.* External morphology model can be created easily using

```
encode_dictionary external unknown_tag >output_model
```

No standard input is read in this case. The `unknown_tag` parameter is used when no tag is assigned to a word form during analysis. The resulting model is printed on standard output.

The external morphology model does not contain any morphological dictionary. Instead, it expects the user to perform morphological analysis and generation on their own. Therefore, the input form to analysis is expected to be followed by space separated lemma-tag pairs, which are returned by the analysis. Similarly, the input lemma to generation is expected to be followed by space separated form-tag pairs, which are again returned by the generation (possibly filtered by a tag wildcard). (To extract the length of the form or lemma itself even when followed by external analyses, API calls `raw_form_len` or `raw_lemma_len` and `lemma_id_len` can be used.)

Note that the tokenizer returned by the external morphology model is the same as the tokenizer of the generic model, and splits input on spaces. Therefore, it can be used to tokenize input, the tokens then passed to the external morphology, and the results can be after proper formatting used as input to MorphoDiTa in vertical input format.

Example input form for analysis using external morphology model:

```
wishes wish NNS wish VBZ
```

Example input lemma for generation using external morphology model:

```
go go VB go VBP goes VBZ going VBG went VBG
```

5.7.2 Custom Tagging Models

Custom tagging models can be trained using `train_tagger` binary, which has the following options:

```
train_tagger generic_234 morphology use_guesser features iterations prune_features
    [heldout_data [early_stopping]] <input_data >tagger_model
```

- `generic_234`: This parameter defines the tagger (elementary features and algorithm) and the order of Viterbi decoding. Use either `generic2`, `generic3` or `generic4`. If unsure, use `generic3` (best released Czech and English models use `generic3`). The `generic2` produces faster, but less accurate models, `generic4` produces larger and only marginally better models.

- `morphology`: File with the morphological dictionary to use.
- `use_guesser`: Use `0/1` to specify whether morphological guesser should be used. Unless you have a good reason not to, use `1`.
- `features`: File with feature sequences for the tagger. The file format and available elementary features are described in following section.
- `iterations`: Number of training iterations. For English, values 5-10 are used, for Czech, values 10-15 are used. Can be affected by `early_stopping`.
- `prune_features`: Use `0/1` to disable/enable pruning of feature sequences not found in training data. Use `1` for smaller and marginally less accurate models, and `0` for larger and marginally better models. If unsure, use `1` (best released Czech and English models use `1`).
- `heldout_data`: Optional file with heldout data in the same format as input data. If supplied, accuracy is measured on the heldout data after every training iteration.
- `early_stopping`: Optionally use `0/1` to disable/enable early stopping. If early stopping is enabled, the resulting model is not the one after the last training iteration, but the one with best heldout data accuracy.

Example command line (use morphology from `morpho.dict`, features from `features.ft` and no heldout data):

```
train_tagger generic3 morpho.dict 1 features.ft 10 1 <input.data >tagger.model
```

Example command line (use morphology from `morpho.dict`, features from `features.ft` and use heldout data with early stopping):

```
train_tagger generic3 morpho.dict 1 features.ft 15 1 heldout.data 1 <input.data
    >tagger.model
```

See next sections for examples of input data and feature files.

Input Data Format

The input data (and the heldout data) represent a sequence of sentences. Different sentences do not interact in any way. Words of one sentence are stored on consecutive lines, each line containing tab separated triplet `form \t lemma \t tag` in UTF-8 encoding. End of sentence is denoted by an empty line.

Example:

```
Děti    dítě    NNFP1-----A----
pojedou jet-1_^(pohybovat_se,_ne_však_chůzí)    VB-P---3F-AA---
k       k-1     RR--3----------
babičce babička NNFS3-----A----
.       .       Z:-------------

Už      už-1    Db-------------
se      se_^(zvr._zájmeno/částice)      P7-X4----------
těší    těšit_:T        VB-S---3P-AA---
.       .       Z:-------------
```

Feature File Format

The features used in the tagger have major influence on tagging performance. The feature file contains several *feature sequences*, each sequence consisting of several *elementary features*. The elementary features are computed by MorphoDiTa and different tagger models can have a different set of elementary features. Here we describe elementary features of `generic` tagger:

- `Form`: word form
- `Prefix1` .. `Prefix9`: word form prefix of length 1..9 (measured in Unicode characters)

- `Suffix1` .. `Suffix9`: word form suffix of length 1..9 (measured in Unicode characters)
- `Num`: whether the word form contains at least one numbers (Unicode category Number)
- `Cap`: whether the word form contains at least one uppercase or titlecase letter
- `Dash`: whether the word form contains at least one dash (Unicode category 'Punctuation, Dash')
- `Tag`: word form PoS tag
- `Tag1` .. `Tag5`: letter 1..5 of word form PoS tag
- `Lemma`: word form lemma
- `FollowingVerbTag`: PoS tag of a nearest following verb, i.e., a nearest following word form with at least one of the PoS tags starting with `V`
- `FollowingVerbLemma`: lemma of a nearest following verb, i.e., a nearest following word form with at least one of the PoS tags starting with `V`
- `PreviousVerbTag`: PoS tag of a nearest previous verb, i.e., a nearest previous word whose PoS tag (assigned by the tagger) starts with `V`
- `PreviousVerbTag`: lemma of a nearest previous verb, i.e., a nearest previous word whose PoS tag (assigned by the tagger) starts with `V`

The feature file defines *feature sequences* which can be applied to a word form. A feature sequence consists of elementary features assigned to the given form or its neighbours.

Every line in the feature file defines one feature sequence. A feature sequence consists of comma joined space separated pairs of elementary feature and an offset to which does the elementary feature apply (i.e., `Form 0` or `Tag 0,Lemma -1`). The file format is strict and does not allow any additional spaces or commas.

Note that offset of some of the elementary features is affected by the order or Viterbi decoding used. Notably, if Viterbi decoding of order *N* is utilized, `Tag` and `Lemma` can be used inside the decoded window, i.e., only with offsets *-N+1 .. 0*.

For inspiration, we present feature files used for releases Czech and English MorphoDiTa models. Both these feature files are slight modifications of feature files described in the paper Spoustová et al. 2009: Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, Miroslav Spousta. 2009. *Semi-Supervised Training for the Averaged Perceptron POS Tagger.* In Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pages 763-771, Athens, Greece, March. Association for Computational Linguistics.

Feature file for English:
```
Tag 0,Form 0
Tag 0,Prefix1 0
Tag 0,Prefix2 0
Tag 0,Prefix3 0
Tag 0,Prefix4 0
Tag 0,Prefix5 0
Tag 0,Prefix6 0
Tag 0,Prefix7 0
Tag 0,Prefix8 0
Tag 0,Prefix9 0
Tag 0,Suffix1 0
Tag 0,Suffix2 0
Tag 0,Suffix3 0
Tag 0,Suffix4 0
Tag 0,Suffix5 0
Tag 0,Suffix6 0
Tag 0,Suffix7 0
Tag 0,Suffix8 0
Tag 0,Suffix9 0
Tag 0,Num 0
Tag 0,Cap 0
Tag 0,Dash 0
Tag 0,Tag -1
Tag 0,Tag -1,Tag -2
Tag 0,Form -1
Tag 0,Form -2
Tag 0,Form -1,Form -2
Tag 0,Form 1
Tag 0,Form 1,Form 2
Tag 0,Tag1 -1
Tag 0,Lemma -1
Lemma 0,Tag -1
```

Feature file for Czech (note that some feature sequences predict only part of PoS tags trying to overcome data sparseness; `Tag2` is extended PoS, `Tag3` is gender, `Tag5` is case):

```
Tag 0
Tag 0,Tag -1
Tag 0,Tag -1,Tag -2
Tag 0,Tag -2
Tag 0,Form 0
Tag 0,Form 0,Form -1
Tag 0,Form -1
Tag 0,Form -2
Tag 0,PreviousVerbTag 0
Tag 0,PreviousVerbLemma 0
Tag 0,FollowingVerbTag 0
Tag 0,FollowingVerbLemma 0
Tag 0,Lemma -1
Lemma 0,Tag -1
Tag 0,Form 1
Tag2 0,Tag5 0
Tag2 0,Tag5 0,Tag2 -1,Tag5 -1
Tag2 0,Tag5 0,Tag2 -1,Tag5 -1,Tag2 -2,Tag5 -2
Tag5 0
Tag5 0,Tag -1
Tag5 0,Tag -1,Tag -2
Tag5 0,Tag -2
Tag5 0,Form 0
Tag5 0,Form 0,Form -1
Tag5 0,Form -1
Tag5 0,Form -2
Tag5 0,PreviousVerbTag 0
Tag5 0,PreviousVerbLemma 0
Tag5 0,FollowingVerbTag 0
Tag5 0,FollowingVerbLemma 0
Tag5 0,Lemma -1
Tag5 0,Form 1
Tag3 0
Tag3 0,Tag -1
Tag3 0,Tag -1,Tag -2
Tag3 0,Tag -2
Tag3 0,Form 0
Tag3 0,Form 0,Form -1
Tag3 0,Form -1
Tag3 0,Form -2
Tag3 0,PreviousVerbTag 0
Tag3 0,PreviousVerbLemma 0
Tag3 0,FollowingVerbTag 0
Tag3 0,FollowingVerbLemma 0
Tag3 0,Lemma -1
Tag3 0,Form 1
Tag 0,Prefix1 0
Tag 0,Prefix2 0
Tag 0,Prefix3 0
Tag 0,Prefix4 0
Tag 0,Suffix1 0
Tag 0,Suffix2 0
Tag 0,Suffix3 0
Tag 0,Suffix4 0
Tag 0,Num 0
Tag 0,Cap 0
Tag 0,Dash 0
Tag5 0,Suffix1 0
Tag5 0,Suffix2 0
Tag5 0,Suffix3 0
Tag5 0,Suffix4 0
```

Feature file for Czech, Part of Speech only variant:

```
Tag 0
Tag 0,Tag -1
Tag 0,Tag -1,Tag -2
Tag 0,Tag -2
Tag 0,Form 0
Tag 0,Form 0,Form -1
Tag 0,Form -1
Tag 0,Form -2
Tag 0,PreviousVerbTag 0
Tag 0,PreviousVerbLemma 0
Tag 0,FollowingVerbTag 0
Tag 0,FollowingVerbLemma 0
Tag 0,Lemma -1
Lemma 0,Tag -1
Tag 0,Form 1
Tag 0,Prefix1 0
Tag 0,Prefix2 0
Tag 0,Prefix3 0
Tag 0,Prefix4 0
Tag 0,Suffix1 0
Tag 0,Suffix2 0
Tag 0,Suffix3 0
Tag 0,Suffix4 0
Tag 0,Num 0
Tag 0,Cap 0
Tag 0,Dash 0
```

Measuring Tagger Accuracy

Measuring custom tagger accuracy can be performed by running:

```
tagger_accuracy tagger_model <test_data
```

This binary reads input in the same format as `train_tagger`, i.e., tab separated form-lemma-tag triplets, and evaluates the accuracy of the tagger model on the given testing data.

6 MorphoDiTa API Tutorial

The MorphoDiTa API is defined in header `morphodita.h` and resides in `ufal::morphodita` namespace. The easiest way to use MorphoDita is therefore:

```
#include morphodita.h

using namespace ufal::morphodita;
```

6.1 Tagger API

The main access to MorphoDiTa tagger is through class `tagger`. An example of this class usage can be found in program file `run_tagger.cpp`. A typical tagger usage may look like this:

```
#include tagger/tagger.h;

using namespace ufal::morphodita;

//...

// load model to memory and construct tagger
tagger* my_tagger = tagger::load("path_to_model");

if (!t) ...

// create sample input
vector<string> words;
words.push_back("malý");
words.push_back("pes");

vector<string_piece> forms;
for (auto& word : words)
  forms.emplace_back(word)

// intialize output and tag
vector<tagged_lemma> tags;
my_tagger->tag(forms, tags);

// access the output
for (auto& tag : tags)
  printf("%s\t%s\n", tag.lemma.c_str(), tag.tag.c_str());

delete my_tagger;
```

The tagger is constructed by an overloaded factory method with one argument. The constructor either accepts an input stream (`istream&`) with the model or a C string (`const char*`) with a file name of the model. The constructor loads the linguistic model to memory and returns the tagger pointer ready for tagging, returning `NULL` if unsuccessful. If an input stream is used, it is positioned right after the end of the model.

The main tagging method is `tagger::tag`:

```
void tag(const std::vector<string_piece>& forms, std::vector<tagged_lemma>& tags) const;
```

The input is a `std::vector` of `string_piece` which is a structure referencing a string using `const char* str` and `size_t len`.

The `tagger::tag` method returns the tagged output in it's second argument, `std::vector<tagged_lemma>`. The calling procedure must provide a result vector and the tagger assigns the output to this vector. Obviously, the indexes in the output vector correspond to indexes in input vector. `tagged_lemma` has two public members: `std::string lemma` and `std:string tag`, corresponding to predicted lemma and tag, respectively.

6.2 Morphological Dictionary API

The main access to MorphoDiTa morphological dictionary is through class `morpho`. An example of this interface usage can be found in a program file `run_morpho.cpp`.

6.2.1 Dictionary Construction

Similarly to the tagger, MorphoDiTa morphological dictionary is constructed by an overloaded factory method which accepts either an input stream (`istream&`) or a C string `const char*` with the file name of the dictionary. The factory method returns a pointer to morphological dictionary or `NULL` if unsuccessful.

```
#include morpho/morpho.h

using namespace ufal::morphodita;

//...

// load dictionary to memory
morpho* my_morpho = morpho::load("path_to_dictionary");

//...

delete(my_morpho);
```

Another way of obtaining a pointer to morphology dictionary is through an instance of `tagger` class – every tagger has a morphology dictionary, which is available through the method

```
virtual const morpho* get_morpho() const = 0;
```

Please note that you should not delete this pointer as it is owned by the `tagger` class instance.

6.2.2 Morphological Analysis

MorphoDiTa morphological dictionary offers two functionalities: It either *analyzes* the given word, that means it outputs all possible lemma-tag pairs candidates for the given form; or for a given lemma-tag pair, it *generates* a form or a whole list of possible forms.

In the first case, one performs morphological analysis for a given word by calling a method `morpho::analyze`:

```
int analyze(string_piece form, guesser_mode guesser, std::vector<tagged_lemma>& lemmas)
    const;
```

An example (assuming that morphological dictionary is already constructed, see previous example):

```
vector<tagged_lemma> lemmas;    // output

my_morpho->analyze("pes", morpho::GUESSER, vector<tagged_lemma>& lemmas);

for (auto& lemma: lemmas)
  printf ("%s %s\n, lemma.lemma.c_str(), lemma.tag.c_str())
```

The input is a form to analyze, then a Guesser mode (whether to use some kind of guesser or strictly dictionary only, see question Guesser Mode in Questions and Answers) and output `std::vector<tagged_lemma>`. The caller must provide an output vector `std::vector<tagged_lemma>` and the method `morpho::analyze` assigns the output to this vector.

6.2.3 Generation

MorphoDiTa performs morphological generation from a given lemma:

```
int generate(string_piece lemma, const char* tag_wildcard, guesser_mode guesser,
             std::vector<tagged_lemma_forms>& forms) const;
```

Tag Wildcard

Optionally, a tag wildcard can be specified (or be `NULL`) and if so, results are filtered using this wildcard. This method can be therefore used in more ways: One may wish to generate all possible forms and their tags from a given lemma. Then the `tag_wildcard` is set to `NULL` and the method generates all possible combinations. One may also need a generate a specific form and tag from a given lemma, then `tag_wildcard` is set to this tag value.

Or even more, for example, in the Czech positional morphology tagging system (Hajič 2004), one may even wish to generate something like "all forms in fourth case", then `tag_wildcard` should be set to `????4`. Please see Section "Czech Morphology" in User's Manual for more details about the Czech positional tagging system. The previous example applies to morphological annotation of PDT, however, the tag wildcards can be used in any morphological tagging system.

Most characters of a tag wildcard match corresponding characters of a tag, with the following exceptions:
- `?` matches any character of a tag.
- `[chars]` matches any of the characters listed. The dash `-` has no special meaning and if `]` is the first character in `chars`, it is considered as one of the characters and does not end the group.
- `[^chars]` matches any of the characters *not* listed.

Unknown Lemmas

When the lemma is unknown, MorphoDiTa's generation behavior is defined by Guesser mode (see also question Guesser Mode in Questions and Answers). If at least one lemma is found in the dictionary, `NO_GUESSER` is returned. If `guesser == GUESSER` and the lemma is found by the guesser, `GUESSER` is returned. Otherwise, forms are cleared and `-1` is returned.

6.3 Questions and Answers

What is a Guesser Mode?
Morphological analysis may try to guess the lemma and tag of an uknown word. This option is turned on by `morpho::GUESSER` and off by `morpho::NO_GUESSER`.

Why 'string_piece" and not `const char*` or `std::string`?
We aim to make MorphoDiTa interface as effective as possible. Because the input strings may be substrings of larger text or come from different than C++ memory regions, we want to avoid the cost of `\\0` padding or `string` conversion. Nevertheless, both `const char*` and `std::string` can be used instead of a `string_piece` because of existing implicit conversion rules.

7 MorphoDiTa API Reference

The MorphoDiTa API is defined in header `morphodita.h` and resides in `ufal::morphodita` namespace.

The strings used in the MorphoDiTa API are always UTF-8 encoded (except from file paths, whose encoding is system dependent).

7.1 MorphoDiTa Versioning

MorphoDiTa is versioned using Semantic Versioning. Therefore, a version consists of three numbers *major.minor.patch*, optionally followed by a hyphen and pre-release version info, with the following semantics:

- Stable versions have no pre-release version info, development have non-empty pre-release version info.
- Two versions with the same *major.minor* have the same API with the same behaviour, apart from bugs. Therefore, if only *patch* is increased, the new version is only a bug-fix release.

- If two versions *v* and *u* have the same *major*, but *minor(v)* is greater than *minor(u)*, version *v* contains only additions to the API. In other words, the API of *u* is all present in *v* with the same behaviour (once again apart from bugs). It is therefore safe to upgrade to a newer MorphoDiTa version with the same *major*.
- If two versions differ in *major*, their API may differ in any way.

Models created by MorphoDiTa have the same behaviour in all MorphoDiTa versions with same *major*, apart from obvious bugfixes. On the other hand, models created from the same data by different *major.minor* MorphoDiTa versions may have different behaviour.

7.2 Lemma Structure

The lemmas used by MorphoDiTa consist of three parts:

1. *raw lemma*: text form of the lemma. May not uniquely distinguish lemma meanings, lemma use cases etc.
2. *lemma id*: together with raw lemma provide a unique identifier of the lemma, possibly including lemma meanings or use cases.
3. *lemma comments*: additional comments for the given lemma.

These parts are stored in one string and the boundaries between them can be determined by `morpho::raw_lemma_len` and `morpho::lemma_id_len` methods. Analyzer and tagger always return lemma in this structured form. When performing morphological generation, either *raw lemma* or both *raw lemma* and *lemma id* can be specified, any *lemma comments* are ignored.

7.3 Struct string_piece

```
struct string_piece {
  const char* str;
  size_t len;

  string_piece();
  string_piece(const char* str);
  string_piece(const char* str, size_t len);
  string_piece(const std::string& str);
}
```

The `string_piece` is used for efficient string passing. The string referenced in `string_piece` is not owned by it, so users have to make sure the referenced string exists as long as the `string_piece`.

7.4 Struct tagged_form

```
struct tagged_form {
  std::string form;
  std::string tag;
};
```

The `tagged_form` is a pair of strings used when obtaining a form and tag pair.

7.5 Struct tagged_lemma

```
struct tagged_lemma {
  std::string lemma;
  std::string tag;
};
```

The `tagged_lemma` is a pair of strings used when obtaining a lemma and tag pair.

7.6 Struct tagged_lemma_forms

```
struct tagged_lemma_forms {
  std::string lemma;
  std::vector<tagged_form> forms;
};
```

The tagged_lemma_forms represents a lemma and a list of tagged forms.

7.7 Struct token_range

```
struct token_range {
  size_t start;
  size_t length;
};
```

The token_range represent a range of a token as returned by a tokenizer. The start and length fields specify the token position in Unicode characters, not in bytes of UTF-8 encoding.

7.8 Struct derivated_lemma

```
struct derivated_lemma {
  std::string lemma;
};
```

The derivated_lemma structure stores information about a derivation. This information currently consists of lemma only, but a type of the derivation may be added later.

7.9 Class version

```
class version {
 public:
  unsigned major;
  unsigned minor;
  unsigned patch;
  std::string prerelease;

  static version current();
};
```

The version class represents MorphoDiTa version. See MorphoDiTa Versioning for more information.

7.9.1 version::current

```
static version current();
```

Returns current MorphoDiTa version.

7.10 Class tokenizer

```
class tokenizer {
 public:
  virtual ~tokenizer() {}

  virtual void set_text(string_piece text, bool make_copy = false) = 0;
  virtual bool next_sentence(std::vector<string_piece>* forms, std::vector<token_range>*
      tokens) = 0;

  static tokenizer* new_vertical_tokenizer();
  static tokenizer* new_czech_tokenizer();
  static tokenizer* new_english_tokenizer();
  static tokenizer* new_generic_tokenizer();
};
```

The tokenizer class performs segmentation and tokenization of given text. The class is *not* threadsafe.

The tokenizer instances can be obtained either directly using static methods or through instances of morpho and tagger.

7.10.1 tokenizer::set_text

```
virtual void set_text(string_piece text, bool make_copy = false) = 0;
```

Set the text which is to be tokenized.

If make_copy is false, only a reference to the given text is stored and the user has to make sure it exists until the tokenizer is released or set_text is called again. If make_copy is true, a copy of the given text is made and retained until the tokenizer is released or set_text is called again.

7.10.2 tokenizer::next_sentence

```
virtual bool next_sentence(std::vector<string_piece>* forms, std::vector<token_range>*
    tokens) = 0;
```

Locate and return next sentence of the given text. Returns true when successful and false when there are no more sentences in the given text. The arguments are filled with found tokens if not NULL. The forms contain token ranges in bytes of UTF-8 encoding, the tokens contain token ranges in Unicode characters.

7.10.3 tokenizer::new_vertical_tokenizer

```
static tokenizer new_vertical_tokenizer();
```

Returns a new instance of a vertical tokenizer, which considers every line to be one token, with empty line denoting end of sentence. The user should delete the instance after use.

7.10.4 tokenizer::new_czech_tokenizer

```
static tokenizer new_czech_tokenizer();
```

Returns a new instance of a Czech tokenizer. The user should delete it after use.

If two MorphoDiTa versions have the same *major.minor*, this tokenizer should behave identically (apart from obvious bugfixes). Nevertheless, the behaviour of this tokenizer might change in different *major.minor* version. If you need a tokenizer whose behaviour does not change, use tokenizer embedded in a morphological dictionary.

7.10.5 tokenizer::new_english_tokenizer

```
static tokenizer new_english_tokenizer();
```

Returns a new instance of a English tokenizer. The user should delete it after use.

If two MorphoDiTa versions have the same *major.minor*, this tokenizer should behave identically (apart from obvious bugfixes). Nevertheless, the behaviour of this tokenizer might change in different *major.minor* version. If you need a tokenizer whose behaviour does not change, use tokenizer embedded in a morphological dictionary.

7.10.6 tokenizer::new_generic_tokenizer

```
static tokenizer new_generic_tokenizer();
```

Returns a new instance of a generic tokenizer. The user should delete it after use.

If two MorphoDiTa versions have the same *major.minor*, this tokenizer should behave identically (apart from obvious bugfixes). Nevertheless, the behaviour of this tokenizer might change in different *major.minor* version. If you need a tokenizer whose behaviour does not change, use tokenizer embedded in a morphological dictionary.

7.11 Class derivator

```
class derivator {
 public:
  virtual ~derivator();

  virtual bool parent(string_piece lemma, derivated_lemma& parent) const = 0;
  virtual bool children(string_piece lemma, std::vector<derivated_lemma>& children) const =
      0;
};
```

The `derivator` class perform morphological derivation on given lemmas. The derivation are computed using lemma ids, see Lemma Structure.

The `derivator` instances can be obtained through instances of `morpho` (and transitively through `tagger`).

7.11.1 derivator::parent

```
virtual bool parent(string_piece lemma, derivated_lemma& parent) const = 0;
```

Return the parent of a given lemma in the morphological derivation tree. The lemma is assumed to be lemma id (see Lemma Structure), so if it contains any lemma comments, they are ignored.

The returned lemma is a full lemma (lemma id plus appropriate lemma comments).

If no parent exists, the function empties the parent lemma and returns `false`.

7.11.2 derivator::children

```
virtual bool children(string_piece lemma, std::vector<derivated_lemma>& children) const = 0;
```

Return children of a given lemma in the morphological derivation tree. The lemma is assumed to be lemma id (see Lemma Structure), so if it contains any lemma comments, they are ignored.

The returned lemmas are full lemmas (lemma ids plus appropriate lemma comments).

If no children exist, the function empties the children vector and returns `false`.

7.12 Class derivation_formatter

```
class derivation_formatter {
 public:
  virtual ~derivation_formatter() {}

  virtual void format_derivation(std::string& lemma) const = 0;

  static derivation_formatter* new_none_derivation_formatter();
  static derivation_formatter* new_root_derivation_formatter(const derivator* derinet);
  static derivation_formatter* new_path_derivation_formatter(const derivator* derinet);
  static derivation_formatter* new_tree_derivation_formatter(const derivator* derinet);
  static derivation_formatter* new_derivation_formatter(string_piece name, const derivator*
      derinet);
};
```

The derivation_formatter class performs required morphological derivation and formats the results using a single string field (i.e., directly in the lemma).

7.12.1 derivation_formatter::format_derivation

```
virtual void format_derivation(std::string& lemma) const = 0;
```

Perform the required morphological derivation and format the result back directly in the lemma.

7.12.2 derivation_formatter::new_none_derivation_formatter

```
static derivation_formatter* new_none_derivation_formatter();
```

Return a new derivation_formatter instance which does nothing (i.e., it performs no derivation).

7.12.3 derivation_formatter::new_root_derivation_formatter

```
static derivation_formatter* new_root_derivation_formatter(const derivator* derinet);
```

Return a new derivation_formatter instance which replaces a lemma by the corresponding root in the derivation tree.

7.12.4 derivation_formatter::new_path_derivation_formatter

```
static derivation_formatter* new_path_derivation_formatter(const derivator* derinet);
```

Return a new derivation_formatter instance which replaces a lemma by a space separated path to the root in the morphological derivation tree (the original lemma is first, followed by its parent, with the root being the last one).

7.12.5 derivation_formatter::new_tree_derivation_formatter

```
static derivation_formatter* new_tree_derivation_formatter(const derivator* derinet);
```

Return a new derivation_formatter instance which appends to the lemma the whole morphological derivation tree which contains it.

The tree is encoded in the following way: root node is the first, then the subtrees of the root children are encoded recursively (each after one space), followed by a final space (which denotes that the children are complete).

7.12.6 derivation_formatter::new_derivation_formatter

```
static derivation_formatter* new_derivation_formatter(string_piece name, const derivator*
    derinet);
```

Return one of the available derivation_formatter instances according to the name parameter:
- none: return new_none_derivation_formatter instance
- root: return new_root_derivation_formatter instance
- path: return new_path_derivation_formatter instance
- tree: return new_tree_derivation_formatter instance

7.13 Class morpho

```
class morpho {
 public:
  virtual ~morpho() {}

  static morpho* load(const char* fname);
  static morpho* load(istream& is);
  enum guesser_mode { NO_GUESSER = 0, GUESSER = 1 };

  virtual int analyze(string_piece form, guesser_mode guesser, std::vector<tagged_lemma>&
      lemmas) const = 0;
  virtual int generate(string_piece lemma, const char* tag_wildcard, guesser_mode guesser,
      std::vector<tagged_lemma_forms>& forms) const = 0;

  virtual int raw_lemma_len(string_piece lemma) const = 0;
  virtual int lemma_id_len(string_piece lemma) const = 0;
  virtual int raw_form_len(string_piece form) const = 0;

  virtual tokenizer* new_tokenizer() const = 0;

  virtual const derivator* get_derivator() const;
};
```

A `morpho` instance represents a morphological dictionary. Such a dictionary allow morphological analysis, morphological generation provide information about lemma structure and provides a suitable tokenizer. All methods are thread-safe.

7.13.1 morpho::load(const char*)

```
static morpho* load(const char* fname);
```

Factory method constructor. Accepts C string with a file name of the model. Returns a pointer to an instance of `morpho` which the user should delete after use.

7.13.2 morpho::load(istream&)

```
static morpho* load(istream& is);
```

Factory method constructor. Accepts an input stream with the model. Returns a pointer to an instance of `morpho` which the user should delete after use.

7.13.3 morpho::guesser_mode

```
enum guesser_mode { NO_GUESSER = 0, GUESSER = 1 };
```

Guesser mode defines behavior in case of unknown words. When set to `GUESSER`, morpho tries to guess unknown words. When set to `NO_GUESSER`, morpho does not guess unknown words.

7.13.4 morpho::analyze()

```
virtual int analyze(string_piece form, guesser_mode guesser, std::vector<tagged_lemma>&
    lemmas) const = 0;
```

Perform morphological analysis of a form. The guesser parameter specifies whether a guesser can be used if the form is not found in the dictionary. Output is assigned to the lemmas vector.

If the form is found in the dictionary, analyses are assigned to lemmas and `NO_GUESSER` returned. If `guesser == GUESSER` and the form analyses are found using a guesser, they are assigned to lemmas and `GUESSER` is returned. Otherwise `-1` is returned and lemmas are filled with one analysis containing given form as lemma and a tag for unknown word.

7.13.5 morpho::generate()

```
virtual int generate(string_piece lemmma, const char* tag_wildcard, guesser_mode guesser,
    std::vector<tagged_lemma_forms>& forms) const = 0;
```

Perform morphological generation of a lemma. Optionally a tag_wildcard can be specified (or be `NULL`) and if so, results are filtered using this wildcard. The guesser parameter speficies whether a guesser can be used if the lemma is not found in the dictionary. Output is assigned to the forms vector.

Tag_wildcard can be either `NULL` or a wildcard applied to the results. A `?` in the wildcard matches any character, `[bytes]` matches any of the bytes and `[^bytes]` matches any byte different from the specified ones. A `-` has no special meaning inside the bytes and if `]` is first in bytes, it does not end the bytes group.

If the given lemma is only a raw lemma, all lemma ids with this raw lemma are returned. Otherwise only matching lemma ids are returned, ignoring any lemma comments. For every found lemma, matching forms are filtered using the tag_wildcard. If at least one lemma is found in the dictionary, `NO_GUESSER` is returned. If `guesser == GUESSER` and the lemma is found by the guesser, `GUESSER` is returned. Otherwise, forms are cleared and `-1` is returned.

7.13.6 morpho::raw_lemma_len

```
virtual int raw_lemma_len(string_piece lemma) const = 0;
```

When given a lemma returned by the dictionary, returns the length of a *raw lemma* (see Lemma Structure).

7.13.7 morpho::lemma_id_len

```
virtual int lemma_id_len(string_piece lemma) const = 0;
```

When given a lemma returned by the dictionary, returns the length of a *raw lemma* plus a *lemma id* (see Lemma Structure). Therefore, the substring of the original lemma of this length is a unique lemma identifier. The rest of the original lemma are lemma comments which do not identify the lemma.

7.13.8 morpho::raw_form_len

```
virtual int raw_form_len(string_piece form) const = 0;
```

When given a form, returns the length of a *raw form*. This is used only in *external morphology model*, where form contains also morphological analyses, and this call can return the length of the form without the analyses.

7.13.9 morpho::new_tokenizer

```
virtual tokenizer* new_tokenizer() const = 0;
```

Returns a new instance of a suitable tokenizer or `NULL` if no such tokenizer exists. The user should delete it after use.

Note that the tokenizer might use the `morpho` instance, so the tokenizer must not be used after the `morpho` instance is destructed.

7.13.10 morpho::get_derivator

```
virtual const derivator* get_derivator() const;
```

Returns a `derivator` for the morphology, or `NULL` if not available.

The `derivator` is owned by the morphology, so the returned instance should not be freed and it cannot be used after the `morpho` instance is destructed.

7.14 Class tagger

```
class tagger {
 public:
  virtual ~tagger() {}

  static tagger* load(const char* fname);
  static tagger* load(istream& is);

  virtual const morpho* get_morpho() const = 0;

  virtual void tag(const std::vector<string_piece>& forms, std::vector<tagged_lemma>& tags,
      morpho::guesser_mode guesser = -1) const = 0;

  virtual void tag_analyzed(const std::vector<string_piece>& forms,
      std::vector<std::vector<tagged_lemma> >& analyses, std::vector<int>& tags) const = 0;

  tokenizer* new_tokenizer() const = 0;
};
```

A `tagger` instance represents a tagger, which perform disambiguation of morphological analyses. All methods are thread-safe.

7.14.1 tagger::load(const char*)

```
static tagger* load(const char* fname);
```

Factory method constructor. Accepts C string with a file name of the model. Returns a pointer to an instance of `tagger` which the user should delete after use.

7.14.2 tagger::load(istream&)

```
static tagger* load(istream& is);
```

Factory method constructor. Accepts an input stream with the model. Returns a pointer to an instance of `tagger` which the user should delete after use.

7.14.3 tagger::get_morpho()

```
virtual const morpho* get_morpho() const = 0;
```

Returns a pointer to an instance of `morpho` associated with the tagger. Do not delete the pointer, it is owned by the tagger instance and deleted in the tagger destructor.

7.14.4 tagger::tag()

```
virtual void tag(const std::vector<string_piece>& forms, std::vector<tagged_lemma>& tags,
    morpho::guesser_mode guesser = -1) const = 0;
```

Perform morphological analysis and subsequent disambiguation. Accepts a `std::vector` of `string_piece` and fills the output vector of `tagged_lemma`.

The ‘guesser‘ parameter defines whether morphological guesser should be used. If negative value is specified (which is the default), the guesser settings employed when the tagger model was trained is used.

7.14.5 tagger::tag_analyzed()

```
virtual void tag_analyzed(const std::vector<string_piece>& forms,
    std::vector<std::vector<tagged_lemma> >& analyses, std::vector<int>& tags) const = 0;
```

Perform morphological disambiguation using given morphological analyses. The indices of chosen analyses are stored in the output vector `tags`.

None of the `analyses` can be empty – in that case, no operation is performed and `tags` is empty. On the other hand, the `analyses` vector can be larger than `forms` – additional entries are ignored in that case.

Note that the tagger was trained with a specific morphology – the more your morphological analyses differ from the original ones, the worse the results will be. One of the usages of `tag_analyzed` is to consider only a subset of morphological analyses.

7.14.6 tagger::new_tokenizer

```
virtual tokenizer* new_tokenizer() const = 0;
```

Returns a new instance of a suitable tokenizer or `NULL` if no such tokenizer exists. The user should delete it after use. The call is equal to `get_morpho()->new_tokenizer()`.

7.15 Class tagset_converter

```
class tagset_converter {
 public:
  virtual ~tagset_converter() {}

  virtual void convert(tagged_lemma& tagged_lemma) const = 0;
  virtual void convert_analyzed(std::vector<tagged_lemma>& tagged_lemmas) const = 0;
  virtual void convert_generated(std::vector<tagged_lemma_forms>& forms) const = 0;

  static tagset_converter* new_identity_converter();
  static tagset_converter* new_pdt_to_conll2009_converter();
  static tagset_converter* new_strip_lemma_comment_converter(const morpho& dictionary);
  static tagset_converter* new_strip_lemma_id_converter(const morpho& dictionary);
};
```

7.15.1 tagset_converter::convert()

```
virtual void convert(tagged_lemma& tagged_lemma) const = 0;
```

Convert the given tagged lemma.

7.15.2 tagset_converter::convert_analyzed()

```
virtual void convert_analyzed(std::vector<tagged_lemma>& tagged_lemmas) const = 0;
```

Convert the given results of morpho::analyze. Apart from calling convert, any repeated entries are removed.

7.15.3 tagset_converter::convert_generated()

```
virtual void convert_generated(std::vector<tagged_lemma_forms>& forms) const = 0;
```

Convert the given results of morpho::generate. Apart from calling convert, any repeated entries are removed.

7.15.4 tagset_converter::new_identity_converter()

```
static tagset_converter* new_identity_converter();
```

Returns a new instance of an identity converter. All convert methods of an identity converter do nothing. The user should delete the instance after use.

7.15.5 tagset_converter::new_pdt_to_conll2009_converter()

```
static tagset_converter* new_pdt_to_conll2009_converter();
```

Returns a new instance of a Czech PDT tag set to CoNLL2009 tag set converter. The user should delete the instance after use.

CoNLL2009 tag set uses two columns for tags – one is a POS and the other one are additional FEATs. Because we have only one tag field, we merge these fields together by using `Pos=?|FEAT`, i.e., the POS is stored as a first FEAT.

7.15.6 tagset_converter::new_strip_lemma_comment_converter()

```
static tagset_converter* new_strip_lemma_comment_converter(const morpho& dictionary);
```

Returns a new instance of a tag set converter stripping lemma comment using the given `morpho` instance, which must remain valid during existence of the tag set converter. The user should delete the tag set converter instance after use.

7.15.7 tagset_converter::new_strip_lemma_id_converter()

```
static tagset_converter* new_strip_lemma_id_converter(const morpho& dictionary);
```

Returns a new instance of a tag set converter stripping lemma id using the given `morpho` instance, which must remain valid during existence of the tag set converter. The user should delete the tag set converter instance after use.

7.16 C++ Bindings API

Bindings for other languages than C++ are created using SWIG from the C++ bindings API, which is a slightly modified version of the native C++ API. Main changes are replacement of `string_piece` type by native strings and removal of methods using `istream`. Here is the C++ bindings API declaration:

7.16.1 Helper Structures

```
typedef vector<int> Indices;

typedef vector<string> Forms;

struct TaggedForm {
  string form;
  string tag;
};
typedef vector<TaggedForm> TaggedForms;

struct TaggedLemma {
  string lemma;
  string tag;
};
typedef vector<TaggedLemma> TaggedLemmas;
typedef vector<TaggedLemmas> Analyses;

struct TaggedLemmaForms {
  string lemma;
  TaggedForms forms;
};
typedef vector<TaggedLemmaForms> TaggedLemmasForms;

struct TokenRange {
  size_t start;
  size_t length;
};
typedef vector<TokenRange> TokenRanges;

struct DerivatedLemma {
  std::string lemma;
};
typedef vector<DerivatedLemma> DerivatedLemmas;
```

7.16.2 Main Classes

```
class Version {
 public:
  unsigned major;
  unsigned minor;
  unsigned patch;
  string prerelease;

  static Version current();
};

class Tokenizer {
 public:
  virtual void setText(const char* text);
  virtual bool nextSentence(Forms* forms, TokenRanges* tokens);

  static Tokenizer* newVerticalTokenizer();
  static Tokenizer* newCzechTokenizer();
  static Tokenizer* newEnglishTokenizer();
  static Tokenizer* newGenericTokenizer();
};

class Derivator {
 public:
  virtual bool parent(const char* lemma, DerivatedLemma& parent) const;
  virtual bool children(const char* lemma, DerivatedLemmas& children) const;
};

class DerivationFormatter {
 public:
  virtual string formatDerivation(const char* lemma) const;

  static DerivationFormatter* newNoneDerivationFormatter();
  static DerivationFormatter* newRootDerivationFormatter(const Derivator* derivator);
  static DerivationFormatter* newPathDerivationFormatter(const Derivator* derivator);
  static DerivationFormatter* newTreeDerivationFormatter(const Derivator* derivator);
  static DerivationFormatter* newDerivationFormatter(const char* name, const Derivator*
      derivator);
};

class Morpho {
 public:
  static Morpho* load(const char* fname);

  enum { NO_GUESSER = 0, GUESSER = 1 };

  virtual int analyze(const char* form, int guesser, TaggedLemmas& lemmas) const;
  virtual int generate(const char* lemma, const char* tag_wildcard, int guesser,
      TaggedLemmasForms& forms) const;
  virtual string rawLemma(const char* lemma) const;
  virtual string lemmaId(const char* lemma) const;
  virtual string rawForm(const char* form) const;

  virtual Tokenizer* newTokenizer() const;

  virtual Derivator* getDerivator() const;
};
```

```
class Tagger {
 public:
  static Tagger* load(const char* fname);

  virtual const Morpho* getMorpho() const;

  virtual void tag(const Forms& forms, TaggedLemmas& tags, int guesser = -1) const;

  virtual void tagAnalyzed(const Forms& forms, const Analyses& analyses, Indices& tags)
      const;

  Tokenizer* newTokenizer() const;
};

class TagsetConverter {
 public:
  static TagsetConverter* newIdentityConverter();
  static TagsetConverter* newPdtToConll2009Converter();
  static TagsetConverter* newStripLemmaCommentConverter(const Morpho& morpho);
  static TagsetConverter* newStripLemmaIdConverter(const Morpho& morpho);

  virtual void convert(TaggedLemma& lemma) const;
  virtual void convertAnalyzed(TaggedLemmas& lemmas) const;
  virtual void convertGenerated(TaggedLemmasForms& forms) const;
};
```

7.17 C# Bindings

MorphoDiTa library bindings is available in the `Ufal.MorphoDiTa` namespace.

The bindings is a straightforward conversion of the `C++` bindings API. The bindings requires native C++ library `libmorphodita_csharp` (called `morphodita_csharp` on Windows).

7.18 Java Bindings

MorphoDiTa library bindings is available in the `cz.cuni.mff.ufal.morphodita` package.

The bindings is a straightforward conversion of the `C++` bindings API. Vectors do not have native Java interface, see `cz.cuni.mff.ufal.morphodita.Forms` class for reference. Also, class members are accessible and modifiable using using `getField` and `setField` wrappers.

The bindings require native C++ library `libmorphodita_java` (called `morphodita_java` on Windows). If the library is found in the current directory, it is used, otherwise standard library search process is used.

7.19 Perl Bindings

MorphoDiTa library bindings is available in the `Ufal::MorphoDiTa` package. The classes can be imported into the current namespace using the `:all` export tag.

The bindings is a straightforward conversion of the `C++` bindings API. Vectors do not have native Perl interface, see `Ufal::MorphoDiTa::Forms` for reference. Static methods and enumerations are available only through the module, not through object instance.

7.20 Python Bindings

MorphoDiTa library bindings is available in the `ufal.morphodita` module.

The bindings is a straightforward conversion of the `C++` bindings API. In Python 2, strings can be both `unicode` and UTF-8 encoded `str`, and the library always produces `unicode`. In Python 3, strings must be only `str`.

8 Contact

Authors:

- Milan Straka, straka@ufal.mff.cuni.cz
- Jana Straková, strakova@ufal.mff.cuni.cz

MorphoDiTa website.

MorphoDiTa LINDAT/CLARIN entry.

9 Acknowledgements

This work has been using language resources developed and/or stored and/or distributed by the LINDAT/CLARIN project of the Ministry of Education of the Czech Republic (project *LM2010013*).

Acknowledgements for individual language models are listed in MorphoDiTa User's Manual.

9.1 Publications

- (Straková et al. 2014) Straková Jana, Straka Milan and Hajič Jan. *Open-Source Tools for Morphology, Lemmatization, POS Tagging and Named Entity Recognition.* In Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations, pages 13-18, Baltimore, Maryland, June 2014. Association for Computational Linguistics.

- (Spoustová et al. 2009) Drahomíra "johanka" Spoustová, Jan Hajič, Jan Raab, Miroslav Spousta. 2009. *Semi-Supervised Training for the Averaged Perceptron POS Tagger.* In Proceedings of the 12th Conference of the European Chapter of the ACL (EACL 2009), pages 763-771, Athens, Greece, March. Association for Computational Linguistics.

9.2 Bibtex for Referencing

```
@InProceedings{strakova14,
  author    = {Strakov\'{a}, Jana  and  Straka, Milan  and  Haji\v{c}, Jan},
  title     = {Open-{S}ource {T}ools for {M}orphology, {L}emmatization, {POS} {T}agging and
      {N}amed {E}ntity {R}ecognition},
  booktitle = {Proceedings of 52nd Annual Meeting of the Association for Computational
      Linguistics: System Demonstrations},
  month     = {June},
  year      = {2014},
  address   = {Baltimore, Maryland},
  publisher = {Association for Computational Linguistics},
  pages     = {13--18},
  url       = {http://www.aclweb.org/anthology/P/P14/P14-5003.pdf}
}
```

9.3 Persistent Identifier

If you prefer to reference MorphoDiTa by a persistent identifier (PID), you can use `http://hdl.handle.net/11858/00-097C-0000-0023-43CD-0`.