# Parsito

Version 1.1.0

# Contents

# 1 Introduction

Parsito is a fast open-source dependency parser written in C++. Parsito is based on greedy transition-based parsing, it has very high accuracy and achieves a throughput of 30K words per second. Parsito can be trained on any input data without feature engineering, because it utilizes artificial neural network classifier. Trained models for all treebanks from Universal Dependencies project are available (37 treebanks as of Dec 2015).

Parsito 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. Parsito 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

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

- Universal Dependencies 1.2 Models: parsito-ud1.2-151120 (documentation)

## 3.2 License

Parsito 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 Straka et al. 2015 and Parsito website.

### 3.3 Platforms and Requirements

Parsito 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 dependency parsing.

## 4 Parsito Installation

Parsito 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 Parsito 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 Parsito, a language model is needed. Here is a list of available language models.

If you want to compile Parsito 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 2.0.5` or newer for language bindings other than `C++`

### 4.2 Compilation

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

Make targets and options:

- `exe`: compile the binaries (default)
- `server`: compile the REST server
- `lib`: compile the static library
- `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

Parsito 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 Parsito binary packages.

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

#### 4.3.2 Java

Binary Java bindings are available in Parsito binary packages.

To compile Java bindings manually, run `make` in the `bindings/java` directory, optionally with the options descriged in Parsito 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-Parsito` package on CPAN.

To compile Perl bindings manually, run `make` in the `bindings/perl` directory, optionally with the options descriged in Parsito 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.parsito` package on PyPI.

To compile Python bindings manually, run `make` in the `bindings/python` directory, optionally with options descriged in Parsito 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 Parsito User's Manual

In a natural language text, the task of dependency parsing is to assign for each word in a sentence its dependency head and dependency relation to the head.

Parsito is a transition-based parser, which greedily chooses transitions from the initial state (all words in a sentence unlinked) to the final state (full dependency tree). It uses an artificial neural network classifier in

every state to choose the next transition to perform. Further details are described in *Straka et al. 2015: Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle.*

Like any supervised machine learning tool, Parsito needs a trained linguistic model. This section describes the available language models and also the commandline tools and interfaces.

## 5.1 Universal Dependencies 1.2 Models

Universal Dependencies 1.2 Models are distributed under the CC BY-NC-SA licence. The models are based solely on Universal Dependencies 1.2 treebanks. The models work in Parsito version 1.0.

Universal Dependencies 1.2 Models are versioned according to the date released in the format `YYMMDD`, where `YY`, `MM` and `DD` are two-digit representation of year, month and day, respectively. The latest version is 151120.

### 5.1.1 Download

The latest version 151120 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 models were trained on Universal Dependencies 1.2 treebanks.

#### Publications

- (Straka et al. 2015) Straka Milan, Hajič Jan, Straková Jana and Hajič Jan jr. *Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle.* In Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories ({TLT\,14}), December 2015.

### 5.1.3 Model Description

The parsing models use the following CoNLL-U fields during parsing:

- `form`
- `upostag`
- `feats`

All other fields (notably `lemma` and `xpostag`) are currently ignored.

Some language models produce non-projective trees and some projective trees, depending on which transition system performed better on development data.

## 5.2 Running the Parser

To run the parser with existing parser model, use
`run_parsito parser_model`

The input is assumed to be in UTF-8 encoding and by default in CoNLL-U format.

Any number of files can be specified after the `parser_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 printed 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_parser` is

```
run_parsito [options] model_file [file[:output_file]]...
Options: --input=conllu
         --output=conllu
         --beam_size=beam size during decoding
         --version
         --help
```

#### 5.2.1 Input Format

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

- `conllu` (default): the CoNLL-U format

#### 5.2.2 Output Format

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

- `conllu` (default): the CoNLL-U format

#### 5.2.3 Beam Search

Optionally, beam search can be used to improve parsing accuracy, at the expense of parsing speed. When using beam search of size *b*, parsing is roughly *1.2 * b* times slower, but the accuracy usually increases.

### 5.3 Running the Parsito REST Server

Parsito also provides REST server binary `parsito_server`. The binary uses MicroRestD as a REST server implementation and provides Parsito REST API.

The full command syntax of `parsito_server` is

```
parsito_server [options] port (model_name model_file acknowledgements beam_size)+
Options: --daemon
         --version
         --help
```

The `parsito_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.

### 5.4 Training Custom Parser Models

Training of Parsito models can be performed using the `train_parsito` binary. The first argument to `train_parsito` is parsing algorithm identifier, currently the only algorithm available is `nn`.

#### 5.4.1 The parsing algorithm nn

The full command syntax of `train_parsito nn` is:

```
train_parsito nn [options] <training_data >parser_model
Options: --adadelta=momentum,epsilon
         --adagrad=learning rate,epsilon
```

```
--batch_size=batch size
--dropout_hidden=hidden layer dropout
--dropout_input=input dropout
--embeddings=embedding description file
--heldout=heldout data file
--hidden_layer=hidden layer size
--hidden_layer_type=cubic|tanh (hidden layer activation function)
--initialization_range=initialization range
--input=conllu (input format)
--iterations=number of training iterations
--l1_regularization=l1 regularization factor
--l2_regularization=l2 regularization factor
--maxnorm_regularization=max-norm regularization factor
--nodes=node selector file
--structured_interval=structured prediction interval
--sgd=learning rate[,final learning rate]
--sgd_momentum=momentum,learning rate[,final learning rate]
--threads=number of training threads
--transition_oracle=static|static_eager|static_lazy|dynamic
--transition_system=projective|swap|link2
--version
--help
```

The required options of `train_parsito nn` are the following. Reasonable defaults are suggested in parentheses:

- `iterations`: number of training iterations to use (`10`)
- `hidden_layer`: size of the hidden layer (`200`)
- `embeddings`: file containing embedding description
- `nodes`: file containing nodes description
- `sgd`, `sgd_momentum`, `adadelta`, `adagrad`: which neural network training algorithm to use (`sgd=0.02,0.001`)
  - `sgd=learning rate[,final learning rate]`: use SGD with specified learning rate, using exponential decay
  - `sgd_momentum=momentum,learning rate[,final learning rate]`: use SGD with momentum and specified learning rate, using exponential decay
  - `adadelta=momentum,epsilon`: use AdaDelta with specified parameters
  - `adagrad=learning rate,epsilon`: use AdaGrad with specified parameters
- `transition_system`: which transition system to use for parsing (language dependant, you can try all and choose the best)
  - `projective`: projective stack-based arc standard system with `shift`, `left_arc` and `right_arc` transitions
  - `swap`: fully non-projective system which extends `projective` system by adding `swap` transition
  - `link2`: partially non-projective system which extends `projective` system by adding `left_arc2` and `right_arc2` transitions
- `transition_oracle`: which transition oracle to use for the chosen `transition_system`:
  - `transition_system=projective`: available oracles are `static` and `dynamic` (`dynamic` usually gives better results, but training time is slower)
  - `transition_system=swap`: available oracles are `static_eager` and `static_lazy` (`static_lazy` almost always gives better results)
  - `transition_system=link2`: only available oracle is `static`

The additional options of `train_parsito nn` are (again with suggested default values):

- `batch_size` (default `1`): use batches of specified size (`10`)
- `dropout_hidden` (default `0`): probability of dropout of hidden layer node
- `dropout_input` (default `0`): probability of dropout of input layer node
- `heldout`: use the specified file as heldout data and report the results of the trained model on them
- `hidden_layer_type` (default `tanh`): hidden layer activation function
  - `tanh`
  - `cubic`
- `initialization_range` (default `0.1`): maximum absolute value of initial random weights in the network
- `input` (default `conllu`): input format to use

- `l1_regularization` (default `0`): L1 regularization
- `l2_regularization` (default `0`): L2 regularization (`0.3`)
- `maxnorm_regularization` (default `0`): if the L2 norm of a row in the network is larger than specified maximum, the row vector is scaled so that its norm is exactly the specified maximum
- `structured_interval` (default `0`): use search-based oracle in addition to the `translation_oracle` specified. This almost always gives better results, but makes training 2-3 times slower. For details, see the paper *Straka et al. 2015: Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle* (use `10` if you want high accuracy and do not mind slower training time)
- `threads` (default 1): if more than 1, train using asynchronous SGD/AdaDelta/AdaGrad with specified number of threads. Note that asynchronous SGD/AdaDelta/AdaGrad is nondeterministic and may give lower results than synchronous one

### Input Format

The input format is specified using the `--input` option. Currently supported input formats are:
- `conllu` (default): the CoNLL-U format

### Embedding description

The embeddings used for every word are specified in the embedding description file. Each line in the file describes one embedding in the following format:

```
embedding_source dimension minimum_frequency [precomputed_embeddings [update_weights
    [maximum_precomputed_embeddings]]]
```

- `embedding_source`: for what data is the embedding created:
  - `form`: word form
  - `lemma`: word lemma
  - `universal_tag`: universal POS tag of the word (the `upostag` field of the input CoNLL-U)
  - `tag`: language-specific POS tag of the word (the `xpostag` field of the input CoNLL-U)
  - `feats`: morphological features of the word (the `feats` field of the input CoNLL-U)
  - `universal_tag_fields`: concatenation of `universal_tag` and `feats`
  - `deprel`: the already assigned dependency relation of the word, of any
- `dimension`: dimension of the embedding
- `minimum_frequency`: only create embeddings for values with the specified minimum frequency. If the minimum frequency is more than 1, embedding for artificial OOV value is created and used for unknown values
- `precomputed_embeddings` (default none): use precomputed embeddings (generated by for example word2vec) from the file specified. The precomputed embeddings file format is the one which `word2vec` uses.
- `update_weights` (default `1`): should the weights of precomputed embeddings be updated further during training:
  - `0`: no, keep the original precomputed embeddings
  - `1`: yes, update the precomputed embeddings
- `maximum_precomputed_embeddings` (default infinity): use at most this many precomputed embeddings (the ones at the beginning of the file are used, which is fine, because the embeddings are usually sorted from the most frequent value)

If unsure, you can use embeddings from *Straka et al. 2015: Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle* (in the paper, embeddings for forms were precomputed using word2vec on the training data):

```
universal_tag 20 1
feats 20 1
form 50 2 [precomputed_embeddings_if_any]
deprel 20 1
```

#### Nodes description

Only some nodes are considered by the classifier in every parser state. Such nodes are specified in the nodes description file, one node per line, in the following format:
`location index[,direction,...]`

The `location` can be one of:
- `stack`: use the stack of processed node, with index `0` representing the node on top of the stack
- `buffer`: use the buffer of not yet processed nodes, with index `0` representing the first node in the buffer

Using `location` and `index`, a node is found. Optionally, its parent or child can be chosen by specifying one or more additional directions in the following format:
- `parent`: choose parent of the current node
- `child,index`: choose a child of the current node, with the first children being 0, 1, 2, ..., and the last children being -3, -2, -1

If unsure, you can use the set of frequently used 18 nodes (used for example by *Zhang and Nivre 2011: Transition-based dependency parsing with rich non-local features*, or *Chen and Manning 2014: A fast and accurate dependency parser using neural networks*, or *Straka et al. 2015: Parsing Universal Dependency treebanks using neural networks and search-based oracle*):
```
stack 0
stack 1
stack 2
buffer 0
buffer 1
buffer 2
stack 0,child 0
stack 0,child 1
stack 0,child -2
stack 0,child -1
stack 1,child 0
stack 1,child 1
stack 1,child -2
stack 1,child -1
stack 0,child 0,child 0
stack 0,child -1,child -1
stack 1,child 0,child 0
stack 1,child -1,child -1
```

### 5.4.2 Measuring Tagger Accuracy

Measuring custom parser model accuracy can be performed by running:
`parsito_accuracy parser_model <test_data`

This binary reads input in the CoNLL-U format containing (probably user-annotated) dependency trees, and evaluates the accuracy of the parser model on the given testing data.

Optionally, beam search can be used to improve parsing accuracy, at the expense of parsing speed. When using beam search of size *b*, parsing is roughly *1.2 * b* times slower, but the accuracy usually increases.

# 6 Parsito API Reference

The Parsito API is defined in header `parsito.h` and resides in `ufal::parsito` namespace. The API allows only using existing models, for custom model creation you have to use the `train_parser` binary.

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

dependent).

## 6.1 Parsito Versioning

Parsito 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 Parsito version with the same *major*.
- If two versions differ in *major*, their API may differ in any way.

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

## 6.2 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`.

## 6.3 Class node

```
class node {
 public:
  int id;              // 0 is root, >0 is sentence node, <0 is undefined
  std::string form;    // form
  std::string lemma;   // lemma
  std::string upostag; // universal part-of-speech tag
  std::string xpostag; // language-specific part-of-speech tag
  std::string feats;   // list of morphological features
  int head;            // head, 0 is root, <0 is without parent
  std::string deprel;  // dependency relation to the head
  std::string deps;    // secondary dependencies
  std::string misc;    // miscellaneous information

  std::vector<int> children;

  node(int id = -1, const std::string& form = std::string())
};
```

The `node` class represents a word in the dependency tree. The `node` fields correspond to CoNLL-U fields, which are documented here, with the `children` field representing the opposite direction of `head` links.

## 6.4 Class tree

```
class tree {
 public:
  tree();

  std::vector<node> nodes;

  bool empty();
  void clear();
  node& add_node(const std::string& form);
  void set_head(int id, int head, const std::string& deprel);
  void unlink_all_nodes();

  static const std::string root_form;
};
```

The `tree` class represents dependency trees of word nodes. Note that the first node (with index 0) is always a technical root, whose form is `root_form`.

Although you can manipulate with the `nodes` directly, the `tree` class offers several simple node manipulation methods.

### 6.4.1 tree::empty()

```
bool empty();
```

Returns `true` if the tree is empty. i.e., if it contains only a technical root node.

### 6.4.2 tree::clear()

```
void clear();
```

Removes all tree nodes but the technical root node.

### 6.4.3 tree::add_node()

```
node& add_node(const std::string& form);
```

Adds a new node to the tree. The new node has first unused `id`, specified `form` and is not linked to any other node. Reference to the new node is returned so that other fields can be also filled.

### 6.4.4 tree:set_head()

```
void set_head(int id, int head, const std::string& deprel);
```

Link the node `id` to the node `head`, with the specified dependency relation. If the `head` is negative, the node `id` is unlinked from its current head, if any.

### 6.4.5 tree::unlink_all_nodes()

```
void unlink_all_nodes();
```

Unlink all nodes.

## 6.5 Class tree_input_format

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

  virtual bool read_block(std::istream& in, std::string& block) const = 0;
```

```
  virtual void set_text(string_piece text, bool make_copy = false) = 0;
  virtual bool next_tree(tree& t) = 0;
  const std::string& last_error() const;

  // Static factory methods
  static tree_input_format* new_input_format(const std::string& name);
  static tree_input_format* new_conllu_input_format();
};
```

The tree_input_format class allows loading dependency trees in various formats.

#### 6.5.1 tree_input_format::read_block()

```
virtual bool read_block(std::istream& in, std::string& block) const = 0;
```

Load from a specified input stream reasonably small text block, which contains complete trees (i.e., the last tree in the block is not incomplete).

Such a text block might be for example a paragraph separated by an empty line.

#### 6.5.2 tree_input_format::set_text()

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

Set the text from which the dependency trees will be read.

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 instance is destroyed or `set_text` is called again. If `make_copy` is `true`, a copy of the given text is made and retained until the instance is destroyed or `set_text` is called again.

#### 6.5.3 tree_input_format::next_tree()

```
virtual bool next_tree(tree& t) = 0;
```

Try reading another dependency tree from the text specified by set_text. Returns `true` if a tree was read and `false` if the text ended of there was a read error.

If the format contains additional information in addition to the fields stored in the tree, it is stored in the tree_input_format instance, and can be printed using a corresponding tree_output_format. Note that this additional information is stored only for the last tree read.

#### 6.5.4 tree_input_format::last_error()

```
const std::string& last_error() const;
```

Returns an error which occurred during the last next_tree. If no error occurred, the returned string is empty.

#### 6.5.5 tree_input_format::new_input_format()

```
static tree_input_format* new_input_format(const std::string& name);
```

Create new tree_input_format instance, given its name. The following input formats are currently supported:

- `conllu`

The new instance must be deleted after use.

#### 6.5.6 tree_input_format::new_conllu_input_format()

```
static tree_input_format* new_conllu_input_format();
```

Creates tree_input_format instance which loads dependency trees in the CoNLL-U format. The new instance must be deleted after use.

Note that even if sentence comments and multi-word tokens are not stored in the tree instance, they can be printed using a corresponding CoNLL-U tree_output_format instance.

## 6.6 Class tree_output_format

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

  virtual void write_tree(const tree& t, std::string& output, const tree_input_format*
      additional_info = nullptr) const = 0;

  // Static factory methods
  static tree_output_format* new_output_format(const std::string& name);
  static tree_output_format* new_conllu_output_format();
};
```

The tree_output_format class allows printing dependency trees in various formats. If the format contains additional information in addition to the fields stored in the tree, it can be printed using a corresponding tree_output_format.

### 6.6.1 tree_output_format::write_tree()

```
virtual void write_tree(const tree& t, std::string& output, const tree_input_format*
    additional_info = nullptr) const = 0;
```

Prints a dependency tree to the specified string.

If the tree was read using a tree_input_format instance, this instance may store additional information, which may be printed by the tree_output_format instance. Note that this additional information is stored only for the tree last read with tree_input_format::next_tree.

### 6.6.2 tree_output_format::new_output_format()

```
static tree_output_format* new_output_format(const std::string& name);
```

Create new tree_output_format instance, given its name. The following output formats are currently supported:

- conllu

The new instance must be deleted after use.

### 6.6.3 tree_output_format::new_conllu_output_format()

```
static tree_output_format* new_conllu_output_format();
```

Creates tree_output_format instance which loads dependency trees in the CoNLL-U format. The new instance must be deleted after use.

Note that even if sentence comments and multi-word tokens are not stored in the tree instance, they can be printed using this instance.

## 6.7 Class parser

```
class parser {
 public:
  virtual ~parser() {};

  virtual void parse(tree& t, unsigned beam_size = 0) const = 0;

  enum { NO_CACHE = 0, FULL_CACHE = 2147483647};
  static parser* load(const char* file, unsigned cache = 1000);
  static parser* load(std::istream& in, unsigned cache = 1000);
};
```

The parser class allows parsing given sentence, using an existing parser model.

### 6.7.1 parser::parse()

```
virtual void parse(tree& t, unsigned beam_size = 0) const = 0;
```

Parses the sentence (passed in the tree instance) and returns a dependency tree. If there are any links in the input tree, they are discarded using tree::unlink_all_nodes first.

The beam size of the decoding can optionally be specified, with the value 0 representing parser model default. If the parser model does not support beam search, the argument is ignored.

### 6.7.2 parser::load(const char*)

```
static parser* load(const char* file, unsigned cache = 1000);
```

Loads parser model from a specified file. Returns a pointer to a new instance of parser which must be deleted after use.

The cache argument specifies caching level, with NO_CACHE representing no caching and FULL_CACHE maximum caching. Although the interpretation of this argument depends on the parser used, you can consider it as a number of most frequent forms/lemmas/tags to cache (either during model loading or during parsing).

### 6.7.3 parser::load(istream&)

```
static parser* load(std::istream& in, unsigned cache = 1000);
```

Loads parser model from the given input stream. The input stream is not closed after loading. Returns a pointer to a new instance of [parser #parser] which must be deleted after use.

The cache argument specifies caching level, with NO_CACHE representing no caching and FULL_CACHE maximum caching. Although the interpretation of this argument depends on the parser used, you can consider it as a number of most frequent forms/lemmas/tags to cache (either during model loading or during parsing).

## 6.8 Class version

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

  static version current();
};
```

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

### 6.8.1 version::current

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

Returns current Parsito version.

## 6.9 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:

### 6.9.1 Helper Structures

```
typedef vector<int> Children;

class Node {
 public:
  int id;          // 0 is root, >0 is sentence node, <0 is undefined
  string form;    // form
  string lemma;   // lemma
  string upostag; // universal part-of-speech tag
  string xpostag; // language-specific part-of-speech tag
  string feats;   // list of morphological features
  int head;       // head, 0 is root, <0 is without parent
  string deprel;  // dependency relation to the head
  string deps;    // secondary dependencies
  string misc;    // miscellaneous information

  Children children;

  node(int id = -1, string form = string());
};
typedef std::vector<node> Nodes;
```

### 6.9.2 Main Classes

```
class Tree {
 public:
  Tree();

  Nodes nodes;

  bool empty();
  void clear();
  node& addNode(string form);
  void setHead(int id, int head, string deprel);
  void unlinkAllNodes();

  static const std::string root_form;
}

class TreeInputFormat {
 public:
  virtual void setText(string text);
  virtual bool nextTree(tree& t) = 0;
  string lastError() const;

  // Static factory methods
  static TreeInputFormat* newInputFormat(string name);
  static TreeInputFormat* newConlluInputFormat();
```

```
};

class TreeOutputFormat {
 public:

  virtual string writeTree(const tree& t, const tree_input_format* additional_info =
      nullptr);

  // Static factory methods
  static TreeOutputFormat* newOutputFormat(string name);
  static TreeOutputFormat* newConlluOutputFormat();
};

class Parser {
 public:
  virtual void parse(tree& t, unsigned beam_size = 0) const;

  enum { NO_CACHE = 0, FULL_CACHE = 2147483647};
  static Parser* load(string file, unsigned cache = 1000);
};

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

  static Version current();
};
```

## 6.10 C# Bindings

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

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

## 6.11 Java Bindings

Parsito library bindings is available in the `cz.cuni.mff.ufal.parsito` package.

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

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

## 6.12 Perl Bindings

Parsito library bindings is available in the `Ufal::Parsito` 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::Parsito::Children` for reference. Static methods and enumerations are available only through the module, not through object instance.

### 6.13 Python Bindings

Parsito library bindings is available in the `ufal.parsito` 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`.

## 7 Contact

Authors:
- Milan Straka, straka@ufal.mff.cuni.cz

Parsito website.

Parsito LINDAT/CLARIN entry.

## 8 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 Parsito User's Manual.

### 8.1 Publications

- (Straka et al. 2015) Straka Milan, Hajič Jan, Straková Jana and Hajič Jan jr. *Parsing Universal Dependency Treebanks using Neural Networks and Search-Based Oracle.* In Proceedings of the Fourteenth International Workshop on Treebanks and Linguistic Theories ({TLT\,14}), December 2015.

### 8.2 Bibtex for Referencing

```
@InProceedings{udparsing:2015,
  author    = {Straka, Milan and Haji\v{c}, Jan and Strakov\'{a}, Jana and Haji\v{c} jr.,
      Jan},
  title     = {Parsing Universal Dependency Treebanks using Neural Networks and
      Search-Based Oracle},
  booktitle = {Proceedings of Fourteenth International Workshop on Treebanks and Linguistic
      Theories ({TLT\,14})},
  month     = {December},
  year      = {2015},
}
```

### 8.3 Permanent Identifier

If you prefer to reference Parsito by a permanent identifier (PID), you can use `http://hdl.handle.net/11234/1-1584`.