ucto Documentation

Language Machines

Jul 18, 2019

Contents

Centre for Language Studies

Radboud University Nijmegen

URL: https://languagemachines.github.io/ucto/

Tokenisation is a process in which text is segmented into the various sentence and word tokens that constitute the text. Most notably, words are separated from any punctuation attached and sentence boundaries are detected. Tokenisation is a common and necessary pre-processing step for almost any Natural Language Processing task, and precedes further processing such as part-of-speech tagging, lemmatisation or syntactic parsing.

Whilst tokenisation may at first seem a trivial problem, it does pose various challenges. For instance, the detection of sentence boundaries is complicated by the usage of periods in abbreviations and the use of capital letters in proper names. Furthermore, tokens may be contracted in constructions such as “I’m”, “you’re”, “father’s”. A tokeniser will generally split these.

UCTO is an advanced rule-based tokeniser. The tokenisation rules used by UCTO are implemented as regular expressions and read from external configuration files, making UCTO flexible and extensible. Configuration files can be further customised for specific needs and for languages not yet supported. Tokenisation rules have first been developed for Dutch, but configurations for several other languages are also provided. UCTO features full unicode support. UCTO is not just a standalone program providing a command-line interface, but is also a C++ library that you can use in your own software. A separate Python binding is also available for use in Python software.

This reference guide is structured as follows. In Chapter [license] you can find the terms of the license according to which you are allowed to use, copy, and modify UCTO. The subsequent chapter gives instructions on how to install the software on your computer. Next, Chapter [implementation] describes the underlying implementation of the software. Chapter [usage] explains the usage.

License and citation

In publication of research that makes use of the Software, a citation should be given of:

Maarten van Gompel, Ko van der Sloot, Iris Hendrickx and Antal van den Bosch. UCTO: Unicode Tokeniser. Reference Guide, Language and Speech Technology Technical Report Series 18-01, Radboud University, Nijmegen, October 30, 2018, Available from https://UCTO.readthedocs.io/.

For information about commercial licenses for the Software, contact lamasoftware@science.ru.nl, or send your request to:

Prof. dr. Antal van den Bosch

Radboud University

P.O. Box 9103 – 6500 HD Nijmegen

The Netherlands

Email: a.vandenbosch@let.ru.nl

CHAPTER 1

Installation on Linux

UCTO can be used as a commandline tool. The UCTO source files can be obtained from:

https://github.com/LanguageMachines/ucto

These sources need to be compiled for the software to run.

However, on most recent Debian and Ubuntu systems, UCTO can be found in the respective software repositories and can be installed with a simple:

```
$ apt-get install ucto
```

On Arch Linux, UCTO is available from the Arch User Repository. Such packages may be behind a bit, however, and not correspond to the latest version.

To facilitate installation of the latest stable UCTO version in most situations, we recommend to use our LaMachine software distribution, which includes UCTO and all dependencies:

http://proycon.github.io/LaMachine/

1.1 Compilation from source

If you, however, install from the source archive, the compilation and installation should also be relatively straightforward on most UNIX systems, and will be explained in the remainder of this section.

To install UCTO, the following libraries need to be installed first:

- `libicu` library, available at http://site.icu-project.org/ but present in the package manager of all major Linux distributions.
- `uctodata`
- `libfolia`, available from `http://proycon.github.com/folia`), which in turn depends on
- *libticcutils*‘ (available from `http://github.com/LanguageMachines/ticcutils`).

After all these dependencies are installed, you can compile UCTO on your computer by running the following from the UCTO source directory:

> $ bash bootstrap.sh $ ./configure

Note: It is possible to install UCTO in a different location than the global default using the `-prefix=<dir>` option, but this tends to make further operations (such as compiling higher-level packages like Frog[1]) more complicated. Use the `-with-UCTO=` option in configure.

After configure you can compile UCTO:

```
$ make
```

and install:

```
$ make install
```

If the process was completed successfully, you should now have an executable file named UCTO in the installation directory (`/usr/local/bin` by default, we will assume this in the remainder of this section), and a dynamic library libUCTO.so in the library directory (`/usr/local/lib/`). The configuration files for the tokeniser can be found in `/usr/local/share/UCTO/`.

UCTO should now be ready for use. You can close and reopen your commandline terminal and issue the UCTO command to verify this. If not found, you may need to add the installation directory (`/usr/local/bin`) to your `$PATH`.

That's all!

The e-mail address for problems with the installation, bug reports, comments and questions is lamasoftware@science.ru.nl.

[1] https://languagemachines.github.io/Frog

CHAPTER 2

Getting Started

UCTO is a regular-expression-based tokeniser offering a command-line interface. The tokeniser program uses a list of regular expressions (rules) and a specified ordering of these regular expresssions to process a text. Each of the rules has a name that signals its purpose like YEAR-NAME, SMILE, ABBREVIATION, etc.

The tokeniser will first split on the spaces already present in the input, resulting in various *fragments*. Each fragment is then matched against the ordered set of regular expressions, until a match is found. If a match is found, the matching part is a token and is assigned the name of the matching regular expression. The matching part may be only a substring of the fragment, in which case there are one or two remaining parts on the left and/or right side of the match. These will be treated as any other fragments and all regular expressions are again tested in the specified order, from the start, and in exactly the same way. This process continues until all fragments have been processed.

Every fragment in the text has been treated after this iterative process and has been labelled with at least one rule name. As a next step, UCTO performs sentence segmentation by looking at a specified list of end-of-sentence markers. Special treatment is given to the period ("."), because of its common use in abbreviations. UCTO will attempt to use capitalisation (for scripts that distinguish case) and sentence length cues to determine whether a period is an actual end of sentence marker or not.

Simple paragraph detection is available in UCTO: a double newline triggers a paragraph break.

Quote detection is also available, but still experimental and by default disabled as it quickly fails on input that is not well prepared. If your input can be trusted on quotes being paired, you can try to enable it. Note that quotes spanning over paragraphs are not supported.

UCTO has a generic configuration file called `generic` that is mostly language independent. We advise to use a language-specific configuration when possible. We offer configuration files for the following languages and the configuration file name should given after `UCTO -L`:

language	-L flag
Dutch	nld
German	deu
English	eng
French	fra
Frysian	fry
Italian	ita
Portuguese	por
Russian	rus
Spanish	spa
Swedish	swe
Turkish	tur

For Dutch we have made specific configuration files for certain domains:

- nld: the default configuration file for Dutch and most up to date and detailed configuration file for Dutch
- nld-historical: a configuration file for historical text, which is more inclined to keep certain punctuation attached to words. Developed in the Nederlab project[3]
- nld-twitter: configuration for Dutch tweets, here the typical URL, email, emoticon and smiley regular expressions are first in order of application
- nld-sonarchat: similar to the nld-twitter configuration but has an additional rule `NICKNAME` to identify the nicknames of authors in a chatroom
- nld-withplaceholder: a 'placeholder' regular expression is the first rule that is applied. The placeholder can be used to prevent certain strings that are marked between % from being changed by the tokenizer.

[3] Nederlab: http://www.nederlab.nl

CHAPTER 3

Usage

UCTO is a command-line tool. The following options are available:

```
Usage:
    UCTO [[options]] [input-file] [[output-file]]
Options:
-c <configfile>   - Explicitly specify a configuration file
    -d <value>        - Set debug level (numerical value 1 or 2)
    -e <string>       - Set input encoding (default UTF8)
    -N <string>       - Set output normalization (default NFC [#f4]_ )
    --filter=[YES|NO] - Disable filtering of special characters
    -f                - OBSOLETE. use --filter=NO
    -h or --help      - This list of options
    -L <language>     - Automatically selects a configuration file by language code.
                      - Available Languages:
                        deu,eng,fra,fry,generic,ita,nld,nld-historical,nld-sonarchat,
↪nld-twitter,nld-withplaceholder,por,rus,spa,swe,tur,
    -l                - Convert to all lowercase
    -u                - Convert to all uppercase
    -n                - One sentence per line (output)
    -m                - One sentence per line (input)
    -v                - Verbose mode
    -s <string>       - End-of-Sentence marker (default: <utt>)
    --passthru        - Don't tokenize, but perform input decoding and simple token␣
↪role detection
    --normalize=<class1>,class2>,...
                      - For class1, class2, etc. output the class tokens instead of␣
↪the tokens itself.
    -T or --textredundancy=[full|minimal|none]  - Set text redundancy level for text␣
↪nodes in FoLiA output:
                        'full'    - Add text to all levels: <p> <s> <w> etc.
                        'minimal' - Don't introduce text on higher levels, but retain␣
↪what is already there.
                        'none'     - Only introduce text on <w>, AND remove all text␣
↪from higher levels
```

(continues on next page)

(continued from previous page)

```
    --filterpunct      - Remove all punctuation from the output
    --uselanguages=<lang1,lang2,..langn> - Only tokenize strings in these languages.␣
↪Default = 'lang1'
    --detectlanguages=<lang1,lang2,..langn> - Try to assign languages before using.␣
↪Default = 'lang1'
    -P                  - Disable paragraph detection
    -Q                  - Enable quote detection (experimental)
    -V or --version     - Show version information
    -x <DocID>          - Output FoLiA XML, use the specified Document ID (obsolete)
    -F                  - Input file is in FoLiA XML. All untokenised sentences will be␣
↪tokenised
                          -F is automatically set when inputfile has extension '.xml'
    -X                  - Output FoLiA XML, use the Document ID specified with --id=
    --id <DocID>        - Use the specified Document ID to label the FoLia doc
                      -X is automatically set when inputfile has extension '.xml'
    --inputclass <class>  - Use the specified class to search text in the FoLia doc.
↪(default is 'current')
    --outputclass <class> - Use the specified class to output text in the FoLia doc.␣
↪(default is 'current')
    --textclass <class>   - Use the specified class for both input and output of text␣
↪in the FoLia doc. (default is 'current'). Implies --filter=NO.
                        (-x and -F disable usage of most other options: -nPQVsS)
```

3.1 Input/output

UCTO has two input formats. It can take either be applied to an untokenised plain text in UTF-8 character encoding as input, or a FoLiA XML document with untokenised sentences. If the latter is the case, the `-F` flag should be added. UCTO will output by default to standard error output in a simplistic format which will simply show all of the tokens and places an `<utt>` symbol where sentence boundaries are detected. If the input text already has sentence boundaries in them, the option “-s ‘‘ followed by the end-sentence-marker string can be used to let UCTO preserve these end-of-sentence-markers.

When UCTO is given two filenames as parameters, the first file will be considdered the input file and the tokenized result will be written to the second file name (and overwrite the content of the second file if it already existed). UCTO will write the output as FoLiA XML when the parameters `-X --id=<filename>` are used.

3.2 Interactive mode

UCTO can also be used in an interactive mode by running the command without specifying an input file. In the interactive mode you type a text (standard input) and the output is given as standard output. This interactive mode is mostly useful when editing a configuration file to adapt the behaviour of UCTO on certain tokens.

3.3 Multilingual text

In case a document consists of mixed multilingual texts, UCTO has an option to apply the automatic language detection tool TextCat[5][6] that guesses the language of a piece of text. UCTO attempts to recognize the language of all fragments

[5] TextCat http://odur.let.rug.nl/vannoord/TextCat/

[6] Cavnar, W. B. and J. M. Trenkle, ‘N-Gram-Based Text Categorization” In Proceedings of Third Annual Symposium on Document Analysis and Information Retrieval, Las Vegas, NV, UNLV Publications/Reprographics, pp. 161-175, 11-13 April 1994. (Available at http://odur.let.rug.nl/vannoord/TextCat/textcat.pdf)

(pieces of text separated by a new line) in the text. UCTO is limited to fragments and cannot handle code switching within a sentence nor recognize the use of one word in one language in a sentence in another language. If you have multiple languages within the same document, you can run UCTO with the option `--detectlanguages=<lang1, lang2,..langn>`. The first language in the specified list will be used as the default language for the whole document. UCTO will first apply TextCat to guess the languages of every fragment in the document. The language-specific configuration will be used on those fragments categorized by TextCat as written in that language for each language that was specified in the list after the `--detectlanguage` parameter. For fragments that were labeled as another (unlisted) language, the first language in the list will be used. Note that the option `--uselanguages` is intended only for Folia XML documents in which the language information was already specified beforehand.

3.4 Example Usage

> Consider the following untokenised input text: *Mr. John Doe goes to the pet store. He sees a cute rabbit, falls in love, and buys it. They live happily ever after.*, and observe the output in the example below.

We save the file to `/tmp/input.txt` and we run UCTO on it. The `-L eng` option sets the language to English and loads the English configuration for UCTO. Instead of `-L`, which is nothing more than a convenient shortcut, we could also use `-c` and point to the full path of the configuration file.

```
$ ucto -L eng /tmp/input.txt
configfile = tokconfig-eng
inputfile = /tmp/input.txt
outputfile =
Initiating tokeniser...
Mr. John Doe goes to the pet store . <utt> He sees a cute rabbit , falls
in love , and buys it . <utt> They live happily ever after . <utt>
```

Alternatively, you can use the `-n` option to output each sentence on a separate line, instead of using the `<utt>` symbol:

```
$ ucto -L eng -n /tmp/input.txt
configfile = tokconfig-eng
inputfile = /tmp/input.txt
outputfile =
Initiating tokeniser...
Mr. John Doe goes to the pet store .
He sees a cute rabbit , falls in love , and buys it .
They live happily ever after .
```

To output to an output file instead of standard output, we would invoke UCTO as follows:

```
$ ucto -L eng /tmp/input.txt /tmp/output.txt
```

This simplest form of output does not show all of the information UCTO has on the tokens. For a more verbose view, add the `-v` option. Now each token is labelled with information about the type of token, and optional functional roles like *BEGINOFSENTENCE* or *NEWPARAGRAPH*. This information can be useful for further NLP processing, and is already used with the Frog NLP pipeline[1].

```
$ ucto -L eng -v /tmp/input.txt
configfile = tokconfig-eng
inputfile = /tmp/input.txt
outputfile =
Initiating tokeniser...
Mr. ABBREVIATION-KNOWN  BEGINOFSENTENCE NEWPARAGRAPH
John    WORD
```

(continues on next page)

(continued from previous page)

```
Doe WORD
goes    WORD
to  WORD
the WORD
pet WORD
store   WORD    NOSPACE
.   PUNCTUATION ENDOFSENTENCE

He  WORD    BEGINOFSENTENCE
sees    WORD
a   WORD
cute    WORD
rabbit  WORD    NOSPACE
,   PUNCTUATION
falls   WORD
in  WORD
love    WORD    NOSPACE
,   PUNCTUATION
and WORD
buys    WORD
it  WORD    NOSPACE
.   PUNCTUATION ENDOFSENTENCE

They    WORD    BEGINOFSENTENCE
live   WORD
happily WORD
ever    WORD
after   WORD    NOSPACE
.   PUNCTUATION ENDOFSENTENCE
```

As you see, this outputs the token types (the matching regular expressions) and roles such as `BEGINOFSENTENCE`, `ENDOFSENTENCE`, `NEWPARAGRAPH`, `BEGINQUOTE`, `ENDQUOTE`, `NOSPACE`. We explain these token types and roles in more detail in the section on Implementation.

For further processing of your file in a natural language processing pipeline, or when releasing a corpus, it is recommended to make use of the FoLiA XML format ###:raw-latex:*cite{FOLIA}*[2]. FoLiA is a format for linguistic annotation supporting a wide variety of annotation types. FoLiA XML output is enabled by specifying the `-X` flag. An ID for the FoLiA document can be specified using the `--id=` flag.

```
$ ucto4 -L eng -v -X --id=example /tmp/input.txt
configfile = tokconfig-eng
inputfile = /tmp/input.txt
outputfile =
Initiating tokeniser...
```

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet type="text/xsl" href="folia.xsl"?>
<FoLiA xmlns:xlink="http://www.w3.org/1999/xlink"
  xmlns="http://ilk.uvt.nl/folia" xml:id="example" generator="libfolia-v0.10">
  <metadata type="native">
    <annotations>
      <token-annotation annotator="ucto" annotatortype="auto" set="tokconfig-en"/>
    </annotations>
  </metadata>
```

(continues on next page)

[2] See also: http://proycon.github.com/folia

(continued from previous page)

```
  <text xml:id="example.text">
    <p xml:id="example.p.1">
      <s xml:id="example.p.1.s.1">
        <w xml:id="example.p.1.s.1.w.1" class="ABBREVIATION-KNOWN">
          <t>Mr.</t>
        </w>
        <w xml:id="example.p.1.s.1.w.2" class="WORD">
          <t>John</t>
        </w>
        <w xml:id="example.p.1.s.1.w.3" class="WORD">
          <t>Doe</t>
        </w>
        <w xml:id="example.p.1.s.1.w.4" class="WORD">
          <t>goes</t>
        </w>
        <w xml:id="example.p.1.s.1.w.5" class="WORD">
          <t>to</t>
        </w>
        <w xml:id="example.p.1.s.1.w.6" class="WORD">
          <t>the</t>
        </w>
        <w xml:id="example.p.1.s.1.w.7" class="WORD">
          <t>pet</t>
        </w>
        <w xml:id="example.p.1.s.1.w.8" class="WORD" space="no">
          <t>store</t>
        </w>
        <w xml:id="example.p.1.s.1.w.9" class="PUNCTUATION">
          <t>.</t>
        </w>
      </s>
      <s xml:id="example.p.1.s.2">
        <w xml:id="example.p.1.s.2.w.1" class="WORD">
          <t>He</t>
        </w>
        <w xml:id="example.p.1.s.2.w.2" class="WORD">
          <t>sees</t>
        </w>
        <w xml:id="example.p.1.s.2.w.3" class="WORD">
          <t>a</t>
        </w>
        <w xml:id="example.p.1.s.2.w.4" class="WORD">
          <t>cute</t>
        </w>
        <w xml:id="example.p.1.s.2.w.5" class="WORD" space="no">
          <t>rabbit</t>
        </w>
        <w xml:id="example.p.1.s.2.w.6" class="PUNCTUATION">
          <t>,</t>
        </w>
        <w xml:id="example.p.1.s.2.w.7" class="WORD">
          <t>falls</t>
        </w>
        <w xml:id="example.p.1.s.2.w.8" class="WORD">
          <t>in</t>
        </w>
        <w xml:id="example.p.1.s.2.w.9" class="WORD" space="no">
```

(continues on next page)

(continued from previous page)

```
          <t>love</t>
        </w>
        <w xml:id="example.p.1.s.2.w.10" class="PUNCTUATION">
          <t>,</t>
        </w>
        <w xml:id="example.p.1.s.2.w.11" class="WORD">
          <t>and</t>
        </w>
        <w xml:id="example.p.1.s.2.w.12" class="WORD">
          <t>buys</t>
        </w>
        <w xml:id="example.p.1.s.2.w.13" class="WORD" space="no">
          <t>it</t>
        </w>
        <w xml:id="example.p.1.s.2.w.14" class="PUNCTUATION">
          <t>.</t>
        </w>
      </s>
      <s xml:id="example.p.1.s.3">
        <w xml:id="example.p.1.s.3.w.1" class="WORD">
          <t>They</t>
        </w>
        <w xml:id="example.p.1.s.3.w.2" class="WORD">
          <t>lived</t>
        </w>
        <w xml:id="example.p.1.s.3.w.3" class="WORD">
          <t>happily</t>
        </w>
        <w xml:id="example.p.1.s.3.w.4" class="WORD">
          <t>ever</t>
        </w>
        <w xml:id="example.p.1.s.3.w.5" class="WORD" space="no">
          <t>after</t>
        </w>
        <w xml:id="example.p.1.s.3.w.6" class="PUNCTUATION">
          <t>.</t>
        </w>
      </s>
    </p>
  </text>
</FoLiA>
```

UCTO can also take FoLiA XML documents with untokenised sentences as input, using the `-F` option.

3.5 Limitations

UCTO simply applies rules to split a text into tokens and sentences. UCTO does not have knowlegde of the meaning of the text and for that reason certain choices will lead to correct tokenisation in most cases but to errors in other cases. An example is the recognition of name initials that prevent a sentence split on names. However, in a example sentence like this, no sentence break will be detected as the 'A.' is seen as a name initial:

- Dutch: *De eerste letter is een A. Dat weet je toch wel.*
- Turkish: *Alfabenin ilk harfi A. Viceversa burada mıydı ?*

Such problematic case cannot be solved by simple rules and would involve more complex solutions such as using word frequency information or using information about the complete text (names tend to re-occur within one text) to determine the likelihood of a word as sentence start. This type of solution goes beyond the current UCTO implementation.

CHAPTER 4

Implementation

The regular expressions on which UCTO relies are read from external configuration files. These configuration files and abbreviation files are stored in the **UCTOdata** git repository at https://github.com/LanguageMachines/uctodata. A configuration file is passed to UCTO using the `-c` or `-L` flags. Several languages have a language-specific configuration file. There are also some separate additional configuration files that contain certain rules that are useful for multiple languages like files for end-of-sentence markers and social media related rules. Configuration files are included for several languages, but it has to be noted that at this time only the Dutch one has been stress-tested to a sufficient extent.

UCTO includes the following separate additional configuration files:

- standard-eos.eos - Standard end-of-sentence markers
- exotic-eos.eos - End-of-sentence markers for more exotic languages.
- smiley.rule - Rules for the detection of smileys/emoticons.
- url.rule - Rules for the detection of URLs.
- email.rule - Rules for the detection of e-mail addresses.

Language-specific abbreviations are listed in a separate file that is referenced in the configuration file as `%include <filename>`. These abbreviation files are created rather ad-hoc, often using https://wiktionary.org as a source for finding language-specific abbreviations.

UCTO uses the unicode character properties and labels specific characters with their unicode property *general category*. Unicode character *symbols* (like the trademark or copyright symbol) are labeled as token type SYMBOL, pictograms like 'thumb up' are labeled as PICTOGRAM, emoji as EMOTICON, currency symbols such as the dollar sign are labeled as CURRENCY.

UCTO starts with dividing a text into fragments based on the spaces already in the text. Next UCTO applies an ordered set of rules to each fragment. Each rule consists of a rule-name and a regular expression. The part of the fragment that matches with the regular expression, is labeled with the rule-name as its token type. For example 'Mr.' matches the rule 'ABBREVIATION-KNOWN' that checks the fragment against the list of known English abbreviations. If the rule only partially matches with the regular expression, the remaining part of the fragment will again be processed using the ordered set of rules until a match is found.

The result of this iterative rule application to all fragments is that all fragments are labeled with their token type. UCTO uses these types to determine the functional roles of ENDOFSENTENCE and BEGINOFSENTENCE.

4.1 Rules

The regular expressions that form the basis of UCTO are defined in *libicu* syntax. This syntax is thoroughly described in the libicu syntax user guide[7] (http://userguide.icu-project.org/strings/regexp).

The configuration file consists of the following sections:

- `RULE-ORDER` – Specifies which rules are included and in what order they are tried. This section takes a space separated list (on one line) of rule identifiers as defined in the `RULES` section. Rules not included here but only in `RULES` will be automatically added to the far end of the chain, which often renders them ineffective.
- `RULES` – Contains the actual rules in format `ID=regexp`, where `ID` is a label identifying the rule, and `regexp` is a regular expression in libicu syntax. The order is specified separately in 'RULE-ORDER', so the order of definition here does not matter.
- `META-RULES` – Contains rules similar to the RULES section but these rules contain an additional placeholder in the rule. The first line of the META-RULES section defines how the placeholder can be recognized. The SPLITTER denotes the special character that will be used to signal the start and end of the placeholder. In most cases the SPLITTER is the % percent sign.
- `ABBREVIATIONS` – Contains a list of known abbreviations, one per line. These may occur with a trailing period in the text, the trailing period is not specified in the configuration. This list will be processed prior to any of the explicit rules. Tokens that match abbreviations from this section get assigned the label `ABBREVIATION-KNOWN`.
- `SUFFIXES` – Contains a list of known suffixes, one per line, that the tokeniser should consider separate tokens. This list will be processed prior to any of the explicit rules. Tokens that match any suffixes in this section receive the label `SUFFIX`.
- `PREFIXES` – Contains a list of known prefixes, one per line, that the tokeniser should consider separate tokens. This list will be processed prior to any of the explicit rules. Tokens that match any suffixes in this section receive the label `PREFIX`.
- `TOKENS` – Treat any of the tokens, one per line, in this list as integral units and do not split. This list will be processed prior to any of the explicit rules. Tokens that match any suffixes in this section receive the label `WORD-TOKEN`.
- `ATTACHEDSUFFIXES` – This section contains suffixes, one per line, that should *not* be separated from the word token to which they are attached. Words containing such suffixes will be marked `WORD-WITHSUFFIX`.
- `ATTACHEDPREFIXES` – This section contains prefixes, one per line, that should *not* be e separated from the word token to which they are attached. Words containing such prefixes will be marked `WORD-WITHPREFIX`.
- `ORDINALS` – Contains suffixes, one per line, used for ordinal numerals. Numbers followed by such a suffix will be marked as `NUMBER-ORDINAL`.
- `UNITS` – This category is reserved for units of measurements, one per line, but is currently disabled due to problems.
- `CURRENCY` – This category is reserved for currency symbols, one per line. The libicu syntax and unicode character encoding already take care of recognizing currency symbols (Sc) like for example $ for US dollars. However, the 3 character currency code strings (like USD, SGD) are not recognized by default. For Dutch we added such codes to the Dutch configuration file.

[7] libicu syntax: http://www.icu-project.org/userguide/regexp

- `EOSMARKERS` – Contains a list of end-of-sentence markers, one per line and in `\uXXXX` format, where `XXXX` is a hexadecimal number indicating a unicode code-point. The period is generally not included in this list as UCTO treats it specially considering its role in abbreviations.
- `QUOTES` – Contains a list of quote-pairs in the format `beginquotes \s endquotes \n`. Multiple begin quotes and end quotes are assumed to be ambiguous.
- `FILTER` – Contains a list of transformations. In the format `pattern \s replacement \n`. Each occurrence of `pattern` will be replaced. This is useful for deconstructing ligatures for example.

Lines starting with a hash sign are treated as comments. Lines starting with `%include` will include the contents of another file. This may be useful if, for example, multiple configurations share many of the same rules, as is often the case.

CHAPTER 5

How to configure UCTO for a new language?

When creating your own configuration, it is recommended to start by copying an existing configuration and use it as an example. We refer to the libicu syntax user guide[7] for the creation of language specific rules. For debugging purposes, run UCTO in a debug mode using `-d <NUMBER>`. The higher the number, the more debug output is produced, showing the exact pattern matching.

Note that the configuration files and abbreviation files are stored in the **UCTOdata** git repository at https://github.com/LanguageMachines/uctodata.

If you created a configuration file for a language or genre that might be useful for a wider audience, please contact us (lamasoftware@science.ru.nl) and we are happy to add your contribution to the main UCTOdata repository.

5.1 Acknowledgments

We thank Ümit Mersinli for his help with the Turkish configuration file.