Ucto: Unicode Tokeniser

version 0.5.3

Reference Guide

ILK Technical Report – ILK 12-05

Maarten van Gompel Ko van der Sloot*
Antal van den Bosch
Centre for Language Studies
Radboud University Nijmegen

(*) Induction of Linguistic Knowledge Research Group
Tilburg Centre for Cognition and Communication
Tilburg University

URL: http://ilk.uvt.nl/ucto/

November 28, 2012

Contents

Introduction

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 preceeds 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 abbreviations and the usage 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 those.

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 English, German, French, Italian, and Swedish are also provided. Ucto features full unicode support. Ucto is not just a standalone program, but is also a C++ library that you can use in your own software.

This reference guide is structured as follows. In Chapter 1 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 3 descibres the underlying implementation of the software. Chapter 4 explains the usage.

Chapter 1

GNU General Public License

In publication of research that makes use of the Software, a citation should be given of: *"Maarten van Gompel, Ko van der Sloot, Antal van den Bosch (2012). Ucto: Unicode Tokeniser. Reference Guide. ILK Technical Report 12-05,*
Available from `http://ilk.uvt.nl/downloads/pub/papers/ilk.1205.pdf` *"*

For information about commercial licenses for the Software, contact `timbl@uvt.nl`, or send your request to:

Prof. dr. Antal van den Bosch
Radboud University Nijmegen
P.O. Box 9103 – 6500 HD Nijmegen
The Netherlands
Email: a.vandenbosch@let.ru.nl

Chapter 2

Installation

You can get the Ucto package as a gzipped tar archive from:

`http://ilk.uvt.nl/ucto`

Following the links from that page, you can download the archive, which contains the complete C++ source code for the program, tokeniser configurations for various language, the license and this documentation. 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
```

If this is the case, you can skip the remainder of this section. 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.

Ucto depends on the `libicu` library. This library can be obtainaed from `http://site.icu-project.org/` but also present in the package manager of all major Linux distributions. It also depends on `libfolia`, available from `http://proycon.github.com/folia`. It will not compile without either of them.

To install the package on your computer, unpack the downloaded file:

```
$ tar -xvzf ucto*.tar.gz
```

This will make a directory `ucto-0.5.1` under your current directory, the precise version number may differ.

Go into this directry and configure the package by typing

```
$ cd ucto-*
```

```
$ ./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 executable file named `ucto` in the installation directory (`/usr/local/bin` by default, we will assume this in the reminder 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/etc/ucto/`.

Ucto should now be ready for use. Reopen your 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 `timbl@uvt.nl`.

[1] `http://ilk.uvt.nl/frog`

Chapter 3

Implementation

Ucto is a regular-expression-based tokeniser. Regular expressions are read from an external configuration file and processed in an order explicitly specified in this same configuration file. Each regular expression has a named label. These labels are propagated to the tokeniser output as tokens processed by a certain regular expression are assigned its identifier.

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 label of the matching regular expression. The matching part may be a 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 are processed.

If a regular expression contains subgroups (marked by parentheses), then not the whole match, but rather the subgroups themselves will become *separate* tokens. Parts within the whole match but not in subgroups are discarded, whilst parts completely outside the match are treated as usual.

Ucto performs sentence segmentation by looking at a specified list of end-of-sentence markers. Whenever an end-of-sentence marker is found, a sentence ends. However, 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.

3.1 Configuration

The regular expressions on which ucto relies are read from external configuration files. A configuration file is passed to ucto using the `-c` or `-L` flags. 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 sufficient extent.

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. This syntax is thoroughly described on `http://userguide.icu-project.org/strings/regexp` . The order is specified seperately in `RULE-ORDER`, so the order of definition here does not matter.
- `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. Libicu regular expression syntax is used again. 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. Libicu regular expression syntax is used again. 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. Libicu regular expression syntax is used again. 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 it. This list will be processed prior to any of the explicit rules. Once more, libicu regular expression syntax is used. 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 split. Words containing such suffixes will be marked `WORD-WITHSUFFIX`.

- `ATTACHEDPREFIXES` – This section contains prefixes, one per line, that should *not* be split. Words containing such prefixes will be marked `WORD-WITHPREFIX`.

- `ORDINALS` – Contains suffixes, one per line, used for ordinal numbers. Number 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, but is currently disabled due to problems.

- `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 endquotes 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. This directive is for the moment only supported within `RULES`, `FILTER`, `QUOTES` and `EOSMARKERS`.

You can see several sections specifying lists. These are implicit regular expressions as all are converted to regular expressions. They are checked prior to any of the explicit rules, in the following order of precedence: `SUFFIXES, PREFIXES, ATTACHEDSUFFIXES, ATTACHEDPREFIXES, TOKENS, ABBREVIATIONS, ORDINALS`.

When creating your own configuration, it is recommended to start by copying an existing configuration and use it as example. For debugging purposes, run ucto in a debug mode using `-d`. The higher the level, the more debug output is produced, showing the exact pattern matching.

Chapter 4

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
-e <string>     - set input encoding (default UTF8)
-N <string>     - set output normalization (default NFC)
-f              - Disable filtering of special characters
-L <language>   - Automatically selects a configuration file
                  by language code
-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
-P              - Disable paragraph detection
-S              - Disable sentence detection!
-Q              - Enable quote detection (experimental)
-V              - Show version information
-F              - Input file is in FoLiA XML. All untokenised
                  sentences will be tokenised.
```

```
-X              - Output FoLiA XML, use the Document ID
                  specified with --id=
--id <DocID>    - use the specified Document ID to label
                  the FoLia doc.
                  (-x and -F disable usage of
                  most other options: -nulPQVsS)
```

Ucto has two input formats and three output formats. It can take either an untokenised plain text UTF-8 as input, or a FoLiA XML document with untokenised sentences. If the latter is the case, the `-F` flag should be added.

Output by default is to standard error output in a simplistic format which will simply show all of the tokens and places a `<utt>` symbol where sentence boundaries are detected. 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 lived 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 en` option sets the language to English and loads the English configuration for ucto. Instead of `-L`, which is nothign more than a convenient shortcut, we could also use `-c` and point to the full path of the configuration file.

```
$ ucto -L en /tmp/input.txt
configfile = tokconfig-en
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 lived 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 en -n /tmp/input.txt
configfile = tokconfig-en
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 lived happily ever after .
```

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

```
$ ucto -L en /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:

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

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 [1] [1]. 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.

```
$ ucto -L en -v -X --id=example /tmp/input.txt
configfile = tokconfig-en
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>
  <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">
```

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

```
      <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">
          <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.

Bibliography

[1] Maarten van Gompel. FoLiA: Format for Linguistic Annotation. Documentation. ILK Technical Report 12-03. available from http://ilk.uvt.nl/downloads/pub/papers/ilk.1203.pdf. ILK Technical Report, 2012.