Sparv 3 - Technical Report

Språkbanken
Institutionen för svenska språket
Göteborgs universitet

CONTENTS

1 **Acknowledgements** 3

2 **The Sparv Pipeline** 4
2.1 Setting up the Sparv Pipeline . . . 4
2.1.1 Requirements . . . 4
2.1.2 Installing the required software . . . 4
2.1.3 Installing additional components (optional) . . . 5
2.1.4 Preparing a corpus . . . 7
2.1.5 Installing Corpus Workbench (optional) . . . 8
2.2 Developing the Sparv Pipeline . . . 9
2.2.1 Downloading and installing the Sparv pipeline . . . 9
2.2.2 The pipeline's folder structure . . . 9
2.2.3 The pipeline structure . . . 9
2.2.4 Annotations . . . 11
2.2.5 Developing a new module . . . 11
2.2.6 Makefile . . . 12
2.3 Sparv's Import Format . . . 12
2.3.1 Format requirements . . . 12
2.3.2 Metadata . . . 13
2.3.3 Configuring the Makefile . . . 13
2.3.4 An example . . . 15
2.4 Parallel Corpora . . . 15
2.4.1 Folder structure . . . 15
2.4.2 Linking corpora . . . 16
2.4.3 The Makefile . . . 16
2.4.4 Annotating . . . 16
2.4.5 Annotation result . . . 17
2.5 Available Sparv Annotations . . . 18

3 **The Sparv Frontend** 20
3.1 Setting up the Sparv Frontend . . . 20
3.1.1 Requirements . . . 20
3.1.2 Installation . . . 20
3.2 Developing the Sparv Frontend . . . 20
3.2.1 Requirements . . . 20
3.2.2 Frontend Configuration . . . 20
3.2.3 Running the Frontend . . . 20

4 **The Sparv Backend** 21
4.1 Setting up the Sparv Backend . . . 21
4.1.1 Info . . . 21
4.1.2 Requirements . . . 21
4.1.3 Installation . . . 21
4.2 Developing the Sparv Backend . . . 21
4.2.1 Backend . . . 22
4.2.2 Catapult . . . 23

5 **Interaction between the Sparv Components** 25

1 ACKNOWLEDGEMENTS

This work and research was supported by Yvonne Adesam, Malin Ahlberg, Gerlof Bouma, Martha Dís Brandt, Peter Ljunglöf, Richard Johansson, Luis Nieto Piña and Roland Schäfer. We thank our colleagues and former colleagues who made great contributions to Sparv.

2 The Sparv Pipeline

2.1 Setting up the Sparv Pipeline

This section describes how to get the Sparv corpus pipeline up and running on your own machine. The source code is available from GitHub[1] under the MIT license[2].

Please note that different license terms may apply to any of the additional components that can be plugged into the Sparv Pipeline!

2.1.1 Requirements

- A Unix-like environment (e.g. Linux, macOS)
- Python 3.4[3] or newer
- GNU Make[4]
- Java[5] (if you want to use the MaltParser, Sparv-wsd or hfst-SweNER)

Additional components (optional):

- Git[6] and Git Large File Storage[7] (for cloning the repository, strongly recommended!)
- Hunpos[8], with its path included in your `PATH` environment variable (for part-of-speech tagging)
- MaltParser[9] v. 1.7.2 (for dependency parsing)
- Sparv-wsd[10] (for word-sense disambiguation)
- hfst-SweNER[11] (for named entity recognition)
- FreeLing[12] 4.1 (if you want to annotate corpora in Catalan, English, French, Galician, German, Italian, Norwegian, Portuguese, Russian, Slovenian or Spanish)
- TreeTagger[13] (if you want to annotate corpora in Bulgarian, Dutch, Estonian, Finnish, Latin, Polish, Romanian or Slovak)
- fast_align[14] (if you want to run word-linking on parallel corpora)
- Corpus Workbench[15] (CWB) 3.2 or newer (if you are going to use the Korp backend for searching in your corpora)

2.1.2 Installing the required software

The following information assumes that you are running Ubuntu, but will most likely work for any Linux-based OS.

1. Install the Sparv Pipeline by cloning the Git repository[16] into a directory of your choice. Please note that you must have Git Large File Storage[17] installed on your machine before cloning the repository. Some files will not be downloaded correctly otherwise. If you do not want to use Git you can download the latest release package from the release page[18].
2. Setup a new environment variable `SPARV_MAKEFILES` and point it to the directory `sparv-pipeline/makefiles`.

[1] https://github.com/spraakbanken/sparv-pipeline
[2] https://opensource.org/licenses/MIT
[3] http://python.org/
[4] https://www.gnu.org/software/make/
[5] http://www.oracle.com/technetwork/java/javase/downloads/jdk8-downloads-2133151.html
[6] https://git-scm.com/
[7] https://git-lfs.github.com/
[8] http://code.google.com/p/hunpos/
[9] http://www.maltparser.org/download.html
[10] https://github.com/spraakbanken/sparv-wsd
[11] http://www.ling.helsinki.fi/users/janiemi/finclarin/ner/hfst-swener-0.9.3.tgz
[12] https://github.com/TALP-UPC/FreeLing/releases/tag/4.1
[13] http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/
[14] https://github.com/clab/fast_align
[15] http://cwb.sourceforge.net/beta.php
[16] https://github.com/spraakbanken/sparv-pipeline
[17] https://git-lfs.github.com/
[18] https://github.com/spraakbanken/sparv-pipeline/releases

2.1.2.1 Python virtual environment

Set up a Python virtual environment as a subdirectory to the `sparv-pipeline` directory:

```
python3 -m venv venv
```

Activate the virtual environment and install the required Python packages (in some cases you are required to upgrade pip first):

```
source venv/bin/activate
pip install --upgrade pip
pip install -r requirements.txt
```

You can then deactivate the virtual environment:

```
deactivate
```

2.1.2.2 Configuring some variables

In `makefiles/Makefile.config` you will find a section called *Configuration*. Here you will need to specify the path to the pipeline directory by setting the variable `SPARV_PIPELINE_PATH`. If you are planning on using the pipeline to install corpora on a remote computer, you must (in addition to installing CWB) edit the `remote_host`, `remote_cwb_datadir` and `remote_cwb_registry` variables. If you are going to use anything database related (generating lemgram index or Word Picture data for Korp) you also have to edit the value of `rel_mysql_dbname`.

2.1.2.3 Downloading and updating the models

Certain annotations need models to work. To download and generate the models follow these simple steps:

1. Navigate to the `sparv-pipeline/models` directory in a terminal.
2. Run the command `make clean`, and then `make all`.
3. If no errors were reported while running the above commands you may run `make space` for saving disk space.

By default, the part-of-speech tagger also relies on the SALDO model. You can disable this dependency by commenting out the line beginning with `hunpos_morphtable` in Makefile.config, or changing its value to `$(SPARV_MODELS)/hunpos.suc.morphtable`.

2.1.3 Installing additional components (optional)

The following components are not part of the core Sparv Pipeline package. Whether or not you will need to install these components depends on how you want to use the Sparv Pipeline. Please note that different licenses may apply for the individual components.

2.1.3.1 Hunpos

Hunpos[19] is used for Swedish part-of-speech tagging and it is a prerequisite for all of the SALDO-annotations. Hunpos can be downloaded from here[20]. Installation is done by unpacking and then adding the path of the executables to your `PATH` environmental variable. If you are running a 64-bit OS, you might also have to install 32-bit compatibility libraries if Hunpos won't run:

```
sudo apt install ia32-libs
```

[19] http://code.google.com/p/hunpos/

[20] http://code.google.com/p/hunpos/

On Arch Linux, activate the `multilib` repository and install `lib32-gcc-libs`.

If that doesn't work, you might have to compile Hunpos from its source code.

2.1.3.2 MaltParser

MaltParser[21] is used for Swedish dependency parsing. The version compatible with the Sparv pipeline is 1.7.2. Download and unpack the zip-file from the MaltParser home page and place its contents under `sparv-pipeline/bin/maltparser-1.7.2`.

Download the dependency model[22] and place the file `swemalt-1.7.2.mco` under `sparv-pipeline/models`.

2.1.3.3 Sparv-wsd

The Sparv-wsd[23] is used for Swedish word-sense disambiguation. It is developed at Språkbanken and runs under the same license as the Sparv Pipeline core package. In order to use it within the Sparv Pipeline it is enough to download the saldowsd.jar from GitHub and place it inside the `sparv-pipeline/bin/wsd` directory:

```
wget https://github.com/spraakbanken/sparv-wsd/raw/master/bin/
   saldowsd.jar -P sparv-pipeline/bin/wsd/
```

Its models are added automatically when building the Sparv Pipeline models (see above).

2.1.3.4 hfst-SweNER

The current version of hfst-SweNER[24] expects to be run in a Python 2 environment while the Sparv pipeline is written in Python 3. If you want to use hfst-SweNER's named entity recognition from within Sparv you need to make sure that your `python` command in your environment refers to Python 2. Alternatively, before installing hfst-SweNER, you can make sure that it will be run with the correct version of Python by replacing `python` with `python2` in all the Python scripts in the `hfst-swener-0.9.3/scripts` directory. The first line in every script will then look like this:

```
#! /usr/bin/env python2
```

2.1.3.5 TreeTagger and FreeLing

TreeTagger[25] and FreeLing[26] are used for POS-tagging and lemmatization of other languages than Swedish. Please install the software according to the instructions on the respective website or in the provided readme file.

The following is a list over the languages currently supported by the corpus pipeline, their language codes and which tool Sparv uses to analyze them:

Language	Code	Analysis Tool
Bulgarian	bg	TreeTagger
Catalan	ca	FreeLing
Dutch	nl	TreeTagger
Estonian	et	TreeTagger
English	en	FreeLing

[21] http://www.maltparser.org/download.html

[22] http://maltparser.org/mco/swedish_parser/swemalt.html

[23] https://github.com/spraakbanken/sparv-wsd

[24] http://www.ling.helsinki.fi/users/janiemi/finclarin/ner/hfst-swener-0.9.3.tgz

[25] http://www.cis.uni-muenchen.de/~schmid/tools/TreeTagger/

[26] https://github.com/TALP-UPC/FreeLing/releases/tag/4.1

Language	Code	Analysis Tool
French	fr	FreeLing
Finnish	fi	TreeTagger
Galician	gl	FreeLing
German	de	FreeLing
Italian	it	FreeLing
Latin	la	TreeTagger
Norwegian	no	FreeLing
Polish	pl	TreeTagger
Portuguese	pt	FreeLing
Romanian	ro	TreeTagger
Russian	ru	FreeLing
Slovak	sk	TreeTagger
Slovenian	sl	FreeLing
Spanish	es	FreeLing
Swedish	sv	Sparv
Swedish 1800's	sv-1800	Sparv
Swedish development mode	sv-dev	Sparv

If you are using TreeTagger, please copy the `tree-tagger` binary file into the `sparv-pipeline/bin/treetagger` directory belonging to the pipeline. The TreeTagger models (parameter files) need to be downloaded separately and saved in the `sparv-pipeline/models/treetagger` directory. The parameter files need to be re-named to a two-letter language code followed by the file ending `.par`, e.g. the Dutch parameter file is called `nl.par`.

When using Freeling you will need the sparv-freeling extension which is available via GitHub[27] and runs under the AGPL license[28]. Follow the installation instructions for the sparv-freeling module on GitHub[29] in order to set up the models correctly.

2.1.3.6 FAST_ALIGN

fast_align[30] is used for word-linking on parallel corpora. Follow the installation instructions given in the fast_align repository and copy the binary files `atools` and `fast_align` into the `sparv-pipeline/bin/word_alignment` folder of the pipeline.

2.1.4 PREPARING A CORPUS

Create a new directory for your corpus (e.g. `mycorpus`). Under your new directory create another directory containing your input texts (e.g. `original`).

The input text must meet the following criteria:

- The files must be encoded in **UTF-8**.
- The format must be **valid XML**, *with the following two exceptions*:
  1. No root element is needed. It is however required that all text in the file must be contained within elements.
     The simplest valid file possible consists of a single element, for example `<text>`, containing nothing but raw text.

  2. Overlapping elements are allowed (`<a><b></a></b>`).
- No file should be larger than **~10-20 MB**, or you might run into memory problems. Split larger files into smaller files.

[27] https://github.com/spraakbanken/sparv-freeling
[28] http://www.gnu.org/licenses/agpl.html
[29] https://github.com/spraakbanken/sparv-freeling
[30] https://github.com/clab/fast_align

Every corpus needs a *Makefile* in which you configure the format of your input material and what annotations you want. You can use the included `Makefile.example` as a base for a simple corpus, or `Makefile.template` if you want to dive into the more advanced stuff. Whichever file you choose, you should put it in the directory you created for your corpus, and name the file `Makefile`.

By default `Makefile.example` is configured to read the source files from a directory named `original`, and assumes that all the text is contained within a `<text>` element. This can easily be changed by editing the `original_dir` and `xml_elements` variables in the Makefile. For a description of every available variable and setting, see `Makefile.template`.

Once you have edited the Makefile to fit your source material and needs, you are ready to execute it by running one of the following commands in a terminal:

`make TEXT`
This will parse your source files, which is a good place to start just to make sure everything works as it should. If there are any problems with the format of your source files, the script will complain.

`make vrt`
This will run every type of annotation that you have specified in the Makefile, and will output a VRT file for every input file. The VRT format is used as input for Corpus Workbench to create binary corpus files. The resulting files will be found in the `annotations` directory (created automatically).

`make export`
Same as `make vrt`, except the output format is XML and the files are saved to the `export.original` directory.

`make cwb`
This command will take the VRT files and convert them to a Corpus Workbench corpus.

`make install_corpus`
This will copy your Corpus Workbench corpus to a remote computer.

`make relations`
Provided that you are using syntactic parsing, this will generate the relations data for the Word Picture in Korp.

`make install_relations`
This will install the relations data to a remote MySQL server.

`make add`
If you remove or add source files to your corpus after having already run one of the commands above, you will have to run the above command.

For a complete list of available commands, run **`make`** without any arguments.

2.1.5 Installing Corpus Workbench (optional)

If you are not going to use the Korp backend, you can skip this step.

You will need the latest version of CWB for unicode support. Install by following these steps:

Check out the latest version of the source code from subversion by running the following command in a terminal:

```
svn co https://cwb.svn.sourceforge.net/svnroot/cwb/cwb/trunk cwb
```

Navigate to the new cwb directory and run the following command:

```
sudo ./install-scripts/cwb-install-ubuntu
```

CWB is now installed, and by default you will find it under `/usr/local/cwb-X.X.X/bin` (where `X.X.X` is the version number). Confirm that the installation was successful by typing:

```
/usr/local/cwb-X.X.X/bin/cqp -v
```

CWB needs two directories for storing the corpora. One for the data, and one for the corpus registry. You may create these directories wherever you want, but let's assume that you have created the following two:

```
/corpora/data
/corpora/registry
```

You also need to edit the following variables in `sparv-pipeline/makefiles/Makefile.config`, pointing to the directories created above:

```
export CWB_DATADIR ?= /corpora/data
export CORPUS_REGISTRY ?= /corpora/registry
```

If you're not running Ubuntu or if you run into any problems, refer to the INSTALL text file in the cwb dir for further instructions.

2.2 Developing the Sparv Pipeline

This section describes the internal structure of the Sparv pipeline and its usage and can be used as a developer's guide.

2.2.1 Downloading and installing the Sparv pipeline

The latest distribution of the pipeline can be downloaded from the installation page[31].

Annotating corpora with the Sparv pipeline can be done in the same manner as on Språkbanken's own servers which is described here[32] (in Swedish only).

The import format is described here[33].

2.2.2 The pipeline's folder structure

```
    annotate/
        bin/        binary files
        makefiles/  Makefiles
        models/     linguistic models
        python/     Python modules
        tests/      tests
```

2.2.3 The pipeline structure

The Sparv corpus pipeline consists of Python modules which are located in the `python` directory. Most of the modules contain code to solve a single task, e.g. part-of-speech tagging, tokenisation etc. Some of the modules call external tools, e.g. Hunpos or the MaltParser.

The following is a short summary of most of the existing modules and their basic functionalities. More information on every script can be found in the doc strings of the functions.

[31] https://spraakbanken.gu.se/en/tools/sparv/pipeline/installation
[32] https://spraakbanken.gu.se/verktyg/sparv/importkedja/korning-av-material
[33] https://spraakbanken.gu.se/en/tools/sparv/pipeline/import-format

Module	Description
annotate.py	Different types of small annotations.
compound.py	Compound analysis using SALDO.
crf.py	Sentence segmentation with Conditional Random Fields.
cwb.py	Creates the different output formats (VRT, XML, Corpus Workbench files).
dateformat.py	Date formatting.
diapivot.py	Creates the diachronic pivot, linking older Swedish texts to SALDO.
fileid.py	Creates IDs for every file in the corpus.
freeling.py	For annotation of other languages. Calls the Freeling software and processes its output. (Only available in the AGPL-version of Sparv)
geo.py	Annotates geographical features.
hist.py	Different functions used for annotation historical Swedish texts.
hunpos_morphtable.py	Creates a morphtable file for Hunpos, based on SALDO.
hunpos.py	Part-of-speech tagging with Hunpos.
info.py	Creates an info file which is used by CWB.
install.py	Installs corpora on other machines.
lemgram_index.py	Creates a lemgramindex which is used by Korp.
lexical_classes.py	Creates annotations for lexical classes with different resources (Blingbring, SweFN).
lmflexicon.py	Parses LMF lexicons.
malt.py	Syntactic parsing with the MaltParser.
number.py	Different types of numbering used e.g. in structural attributes.
parent.py	Adds annotations for parent links and/or children links (needed for pipeline internal purposes).
readability.py	Creates annotations for readability measures.
relations.py	Creates data used by the word picture.
saldo.py	Different types of SALDO-annotations.
segment.py	All kinds of segmentation/tokenisation (paragraph, sentence, word).
sent_align.py	Sentence linking of parallel corpora.
sentiment.py	Creates sentiment annotations.
suc2hunpos.py	Creates a test material based on SUC3 for Hunpos.
swener.py	Creates named-entity annotations with hfst-SweNER.
timespan.py	Handles time spans.
train_nst_comp_model.py	Trains a compound part-of-speech model used by the compound analysis.
train_stats_model.py	Trains a statistical model used by the compound analysis.
treetagger.py	For annotation of other languages. Calls the TreeTagger software and processes its output.
vw.py	Topic modelling with vowpal wabbit.
word_align.py	Word linking of parallel corpora with fast_align and cdec.
wsd.py	Crates annotations for word sense disambiguation.
xmlanalyser.py	An analyzer for pseudo-XML documents.
xmlparser.py	Parses XML and generates files in the pipeline's working format.

The first thing that happens when annotating a corpus with the Sparv pipeline is that the XML input data is parsed by the module xmlparser.py. This module generates the following files for every input file (usually in the `annotations` directory):

- a `.@TEXT` file containing the source text with anchors
- one file for each structural attribute captured, where each line consists of a reference to two anchors indicating the span of the attribute, and the value of the attribute

The output of most modules are files in the above described format, with references to anchors and data for the indicated span. The part-of-speech tagger module for instance creates a file containing an anchor span per word and the part-of-speech as value.

2.2.4 Annotations

Below is a list with some of the most common annotations and which module they are built with:

- *word*
  The string containing a token (word or punctuation). This annotation is created by *xmlparser.py*.
- *msd*
  A morphosyntactic descriptor. Created by *hunpos.py* with *word* as input data.
- *pos*
  A simplified MSD, preserving only its initial part. Created by *annotate.py* with *msd* as input data.
- *baseform, lemgram, sense*
  Citation form, lemgram, and SALDO-ID. Created by *saldo.py*, with *word*, *msd* and *ref* as input files. Uses a SALDO model which is created with *saldo.py*.
- *ref*
  En numrering av varje token inom varje mening. Skapas av *annotate.py*.
- *dephead, deprel*
  Dependency head and dependency relation. Created by *malt.py*, with *word*, *msd* and *pos* as input data. Uses the MALT parser with a model for Swedish.

2.2.5 Developing a new module

This part describes how to create a new Python module within the Sparv pipeline.

Usually one does not have to define the format for the input and output data since there are methods defined for reading and writing these.

Below is a minimal example script which could be integrated in the Sparv pipeline:

```
# -*- coding: utf-8 -*-
import sb.util as util

def changecase(word, out, case):
    # Read the annotation "word"
    words = util.read_annotation(word)

    # Transform every word to upper or lower case
    for w in words:
        if case == "upper":
            words[w] = words[w].upper()
        elif case == "lower":
            words[w] = words[w].lower()

    # Write the result (out) to the indicated annotation (words)
    util.write_annotation(out, words)

if __name__ == '__main__':
    util.run.main(changecase=changecase)
```

Most modules start with importing `util` which contains functions for reading and writing annotations in the data format used within the pipeline.

Then you can define functions that may be used by Sparv. These functions usually take paths to existing annotation files as arguments as well as a path to the annotation file that should be created by the function.

The function `util.read_annotation()` is used to read existing annotations. It returns a dictionary object with anchors as keys and annotation values as values.

A module function can be concluded with calling `util.write_annotation()` which takes a path to an annotation file and a dictionary as arguments and saves the dictionary contents to the specified file path.

A module should also contain some code that makes its functions available from the command line. This is what the last two lines in the above example are there for. As arguments to the function `util.run.main` you can list all the names of the functions you want to be able to use from the command line in the following format: `alias=name_of_function`. The chosen alias can then be specified as a flag in the command line. The first function may be set as default by omitting the alias.

2.2.6 Makefile

When running the Sparv pipeline its different modules are coordinated with makefiles. There are two global makefiles containing configurations and rules for the creation of annotations (`Makefile.config`, `Makefile.rules`). Furthermore, each corpus has its own makefile (`Makefile`) that contains information about the data format and about which annotations should be used. The two global makefiles are imported by the corpus specific makefile.

Documentation for the different corpus specific makefiles can be found in `Makefile.template`. This file is distributed together with the other makefiles in the Sparv pipeline.

`Makefile.rules` is the file containing all rules that create annotations. `Makefile.config` contains variables that can be configured.

A rule in Makefile.rules can for example look like this:

```
    %.token.uppercase: %.token.word
        $(python) -m sb.case --changecase --out $@ --word $(1) --
            case upper
```

The above rule creates the annotation *uppercase* from the Python example above. It takes the annotation *word* as input and calls the function *uppercase* in the script *case.py* with *out*, *word* and *case* as arguments.

Adding a new annotation can in most cases be done by adding a rule like in the above example to `Makefile.rules`. The name of the annotation must also be added to the list of annotations i the corpus specific makefile.

2.3 Sparv's Import Format

This section describes the XML format you should use when importing text using the Sparv corpus pipeline.

2.3.1 Format requirements

The input text files must meet the following criteria:

- The files must be encoded in **UTF-8**.
- The format must be **valid XML**, *with the following two exceptions*:
  1. No root element is needed. It is however required that all text in the file must be contained within elements.
     The simplest valid file possible consists of a single element, for example `<text>`, containing nothing but raw text.

  2. Overlapping elements are allowed (`<a><b></a></b>`).
- No file should be larger than **~10-20 MB**, or you might run into memory problems. Split too large files into smaller files.

2.3.2 Metadata

Any metadata about the text as a whole or parts of the text (e.g. chapters, sentences, words) must exist in the form of attributes to the elements containing the text. For example:

```
<text title="Title metadata here">
  This is the corpus text.
</text>
```

All text, except for element attributes, will be part of the corpus text, and will not be treated as metadata. The following example is *not* a valid way to set title metadata for a text, and the string “Title metadata here” will be treated as corpus text:

```
<text>
  <title>Title metadata here</title>
  This is the corpus text.
</text>
```

2.3.2.1 Headers

One exception to the above is when there is a header in the file, such as:

```
<text>
  <header>
    <title>Title metadata here</title>
  </header>
  This is the corpus text.
</text>
```

In this case it is possible to treat the whole `header` element (not necessarily named `header`) as not belonging to the text, and instead extract metadata from it. The metadata will be tied to all text contained by the root element. In this case the root element is `text`, and the result will be the same as in the first example.

The following is *not* a valid example where you can extract header metadata, due to the header being disconnected from the text:

```
<header>
  <title>Title metadata here</title>
</header>
<text>
  This is the corpus text.
</text>
```

However, had both `header` and `text` been contained within another element, header metadata extraction would have been possible.

2.3.3 Configuring the Makefile

When using the corpus pipeline you configure your Makefile to match your XML format, and tell it what elements and attributes to parse. The variables used for this are the following:

```
xml_elements
xml_annotations
```

In xml_elements you list (in lowercase, separated by whitespace) the elements of interest, optionally with their attributes (on the form element:attribute). xml_annotations is a list of the same size, with every item corresponding to an item in xml_elements. This list contains the names of the annotation files that will be created during parsing. For example:

```
xml_elements =     text:title  text:author  s
xml_annotations = text.title  text.author  sentence
```

The above will parse elements such as <text title="X" author="Y"> and save title and author as separate annotations. It will also parse the attribute <s>, not saving any attribute data but only the boundaries of the element, marking the beginning and end of every sentence (provided that our example text has sentence segmentation marked with <s> tags). In this way you can use existing annotations of a material during the import, rather than letting the pipeline create new annotations for you. By saving the <s> element as the annotation named sentence, the pipeline will not try to create that annotation itself. There are several hardcoded annotation names, all of which can be overridden like sentence above:

```
Positional: token.pos, token.msd, token.baseform, token.lemgram,
   token.saldo, token.prefix, token.suffix
Structural: token, sentence, paragraph
```

For positional annotations (annotations for single tokens), use the prefix “token.” as shown above.

You can combine several elements into one annotation, if for example sentence segmentation is marked up with <s> in some places and <x> in some. Combining them is done by simply joining both elements with a “+”:

```
xml_elements =     s+x
xml_annotations = sentence
```

2.3.3.1 Headers

For files containing headers like in the previously shown example, three variables has to be set in the Makefile:

```
xml_header
xml_headers
xml_header_annotations
```

The first one should be set to the name of the header element, in lowercase. This tells the parser that this element is a header, and none of its text content will be part of the corpus text.

```
xml_header = header
```

Alternatively you can give the full path to the header, joining elements by “.”:

```
xml_header = text.header
```

The second and third variable is similar to the xml_elements/xml_annotations pair. The first one is a list of headers to parse, and the second a list of corresponding annotation files that will be created:

```
xml_headers
xml_header_annotations
```

`xml_headers` must contain the full path to the element in question, in lowercase. As with `xml_elements` you can specify attributes using a colon, but it is also possible to parse the text contained by the element, by using the keyword “TEXT” instead of an attribute:

```
xml_headers            = text.header.title:TEXT
xml_header_annotations = text.title
```

Any periods in element names have to be escaped using backslash.

2.3.4 An example

The following is an example input file, with existing sentence segmentation and tokenization. Every token is annotated with its baseform.

```
<text title="Test">
  <s>
    <w bf="it">It</w>
    <w bf="be">was</w>
    <w bf="an">an</w>
    <w bf="elephant">elephant</w>
    <w bf=".">.</w>
  </s>
</text>
```

The configuration of the Makefile will look like this:

```
xml_elements =    text:title  s         w      w:bf
xml_annotations = text.title  sentence  token  token.baseform
```

2.4 Parallel Corpora

A parallel corpus is a corpus that contains a collection of original texts and their translations into other languages. A parallel corpus can be bilingual or multilingual. In Sparv each text collection of one language is treated as a separate corpus which needs to be annotated separately but may be linked to one or more corpora in other languages. The following example demonstrates how to annotate a bilingual (English-Swedish) parallel corpus with automatic sentence-linking and word-linking.

You can download an example mini corpus including a working Makefile here[34].

2.4.1 Folder structure

Each corpus has its own root directory. The names of the root directories (`mycorpus-en` and `mycorpus-sv`) need to be identical apart from the language ID suffix. If the language in question should be annotated with Sparv its language ID suffix should be a two-letter ISO code (check the language table in the install section for reference). Files that are linked to each other must have the same name (in this example: `sparv-intro.xml`).

[34] https://github.com/spraakbanken/sparv-docs/raw/master/examples/parallel_corpus_example.zip

```
my_parallel_corpus/
    mycorpus-en/
        original/
            xml/
                sparv-intro.xml
    mycorpus-sv/
        original/
            xml/
                sparv-intro.xml
    Makefile
```

2.4.2 Linking corpora

Sparv includes tools for automatic sentence-linking and word-linking. Sentence-linking is a prerequisite for word-linking, and sentence-linking requires pre-aligned text chunks (e.g. paragraphs or similar). In the example corpus the pre-aligned chunks are called “link”. Each link has an ID (attribute `n`) which corresponds to the link-element with the same ID its linked translation.

If you are using automatic sentence-linking you need to set the variable `align_sentences` to `true` and you must specify `sent_align_chunk` to the element that has been pre-aligned.

If your text is already linked on sentence level you need to have `sentlink` elements sourrounding the sentences, in order to get automatic word-linking. They should have IDs indicating which sentences belong together (e.g. `<sentlink n="0001">`). The IDs should be unique for each indata file. Don’t forget to set the variable `align_sentences = false` in your makefile.

If you would like to use automatic word-linking you need to add the attribute `wordlink-$(other_lang)` to the variables `vrt_columns` and `vrt_columns_annotations` in your Makefile. `$(other_lang)` is the variable for the language ID suffix of the linked language.

2.4.3 The Makefile

For parallel corpora the following variables must be set in addition to the standard variables:

- `parallel_base`: the root directory name without language ID suffix (`mycorpus`)
- `root`: the root directory with language ID suffix (`mycorpus-$(lang)`)

For automatic word-linking:

- `align_sentences`: `true` if sentences should be linked, else `false` (In this case the sentences must be pre-aligned. Word-linking will not work without sentence-alignment.)
- `sent_align_chunk`: the pre-aligned chunk which is parent to sentence

You need to include `Makefile.langs` in your Makefile (after including `Makefile.config`). By doing this Sparv will know which analysis tool (TreeTagger or FreeLing) to use for which language.

If the languages you are annotating make use of different analysis modes (e.g. FreeLing, TreeTagger or the standard Swedish mode) you need to define language-specific if-clauses in your Makefile in order for the variables to be set to the correct values. Check the example Makefile for more details.

If you have certain attributes that are not used in all the languages in your parallel corpus you need to list these attributes in the `null_annotations` variable. In that case you also need to set `skip_cwb_compression` and
`skip_cwb_validation` to `true`, otherwise you will not be able to create the Corpus Workbench binaries.

2.4.4 Annotating

Run each corpus separately and define which language should be annotated (`lang`) and which language the corpus should be linked to (`other_lang`; Note: this parameter is set automatically in the example corpus!):

```
make vrt lang=en other_lang=sv
make vrt lang=sv other_lang=en
```

When creating the corpus Workbench binaries you also need to run `make align` for both languages (after creating the vrt-files), e.g.:

```
make align lang=en other_lang=sv
make align lang=sv other_lang=en
```

2.4.5 Annotation result

This is what the vrt result could look like for a Swedish-English parallel corpus (taken from the example corpus which can be downloaded above):

```
<text>
<link n="1">
<sentlink n="sparv-intro.01">
<sentence id="sv0248-sv02a2">
Sparv	NN	NN.UTR.SIN.IND.NOM	|sparv|	|sparv..nn.1|	01	|01|
är	VB	VB.PRS.AKT	|vara|	|vara..vb.1|	02	|02|
Språkbankens	NN	NN.UTR.SIN.DEF.GEN	|språkbank|	|språkbank..nn
    .1|	03	|03|04|
annoteringsverktyg	NN	NN.NEU.PLU.IND.NOM	|	|	04	|05|06|
...
</sentence>
</sentlink>
</link>
</text>
```

```
<text>
<link n="1">
<sentlink n="sparv-intro.01">
<sentence id="en038c2-en0affc">
Sparv	PROPN	NP	sparv	__UNDEF__	01	|01|
is	VERB	VBZ	be	__UNDEF__	02	|02|
an	DET	DT	a	__UNDEF__	03	|03|
annotation	NOUN	NN	annotation	__UNDEF__	04	|03|
tool	NOUN	NN	tool	__UNDEF__	05	|04|
developed	VERB	VBN	develop	__UNDEF__	06	|04|
...
</sentence>
</sentlink>
</link>
</text>
```

The tab-separated columns in the vrt output contain the following information:

`word pos msd baseform lemgram linkref wordlink`

In this example the texts were manually linked by the `<link>` elements. The `sentlink` element was added automatically by the sentence linking module. If the `link` element contains multiple sentences the `sentlink` attribute will break down the linked elements into smaller units. The ID in the `link` and `sentlink` elements (e.g. `n="1"`, `n="sparv-intro.01"`) indicates which elements belong together across linked texts in different languages. By default the `sentlink`-ID is prefixed with the name of the indata file (in this case "sparv-intro"). Note that the prefix is not required for automatic word-linking, so if you use manual sentences-linking you will not need to set a prefix.

The last column (`wordlink`) contains a set of link reference numbers (`linkref`) referring to tokens from the linked text within the same `<link>` element. Note that this set may be empty in some cases, meaning that the token was not linked to any token in the other language.

The column `lemgram` contains only `__UNDEF__` values for the English text because `lemgram` was listed in the `null_annotations` since this annotation is only available for Swedish. Columns containing only `__UNDEF__` values are not present in the XML export.

2.5 Available Sparv Annotations

For texts written in contemporary Swedish Sparv can generate the following types of annotations:

- Part of speech tagging:
  - `pos`: part of speech
  - `msd`: morphosyntactic tag

  Tool: Hunpos[35] Model: in-house model trained on SUC 3.0[36] Tag set: MSD tags[37]
- SALDO[38]-based analysis:
  - `baseform`: citation form
  - `lemgram`: lemgram, identifies the inflectional table (using SALDO tags[39])
  - `sense`: identifies a sense in SALDO and its probability
  - (`saldo`: identifies a sense in SALDO - *will be removed soon*)
  - `sentiment`: sentiment score using SenSALDO[40]
- Compound analysis (also based on SALDO):
  - `complemgram`: compound lemgram
  - `compwf`: compound word form
  - (`prefix`: initial part of a compound - *will be removed soon*)
  - (`suffix`: final part of a compound - *will be removed soon*)
- Dependency analysis:
  - `ref`: the position of the word in the sentence
  - `dephead`: dependency head, the ref of the word which the current word modifies or is dependent of
  - `deprel`: dependency relation, the relation of the current word to its dependency head

  Tool: MaltParser[41] Model: swemalt, trained on Swedish Treebank Tag set: Mamba-Dep[42]
- Named entity recognition:
  - `ne.ex`: named entity (name expression, numerical expression or time expression)
  - `ne.type`: named entity type
  - `ne.subtype`: named entity sub type

  Tool: hfst-SweNER[43] References: HFST-SweNER – A New NER Resource for Swedish[44], Reducing the effect of name explosion[45]
- Readability metrics:
  - `text.lix`: the Swedish readability metric LIX, läsbarhetsindex
  - `text.ovix`: the Swedish readability metric OVIX, ordvariationsindex
  - `text.nk`: the Swedish readability metric nominalkvot
- Lexical classes:

[35] http://code.google.com/p/hunpos/

[36] https://spraakbanken.gu.se/en/resources/suc3

[37] https://spraakbanken.gu.se/korp/markup/msdtags.html

[38] https://spraakbanken.gu.se/en/resources/saldo

[39] https://spraakbanken.gu.se/en/resources/saldo/tagset

[40] https://spraakbanken.gu.se/en/resources/sensaldo

[41] http://www.maltparser.org/download.html

[42] http://stp.ling.uu.se/~nivre/swedish_treebank/dep.html

[43] http://www.ling.helsinki.fi/users/janiemi/finclarin/ner/hfst-swener-0.9.3.tgz

[44] http://www.lrec-conf.org/proceedings/lrec2014/pdf/391_Paper.pdf

[45] http://demo.spraakdata.gu.se/svedk/pbl/kokkinakisBNER.pdf

- `blingbring`: lexical class from the Blingbring resource (on word level)
- `swefn`: frames from swedish FrameNet (on word level)
- `text.blingbring`: lexical class from the Blingbring resource (on document level)
- `text.swefn`: frames from swedish FrameNet (on document level)

Older Swedish texts or texts written in other languages can often be annotated with a sub set of the above annotation types.

The `msd` annotation for non-Swedish languages is based on different tag sets, depending on which language is annotated and what annotation tool is being used. The attribute contains information about the part of speech and in many cases morphosyntactic information. The `pos` annotation contains only part-of-speech information and uses the universal POS tag set[46].

[46] http://universaldependencies.org/u/pos/

3 The Sparv Frontend

3.1 Setting up the Sparv Frontend

This section describes how to get the Sparv frontend up and running on your own machine. Begin by downloading the latest source zip from here[47]. The source code is made available under the MIT license[48].

3.1.1 Requirements

- A web server, such as Apache[49] or equivalent.

3.1.2 Installation

The frontend comes pre-compiled in the zip and can be installed by simply dropping the `dist` folder into your web server root. Consult your web server documentation for more details. You will also need to set some configuration variables in `dist/config.js` in order to point your frontend to the correct backend address and to the address of the default JSON schema file.

3.2 Developing the Sparv Frontend

This section describes the internal structure of the Sparv frontend and can be used as a developer's guide.

The frontend is available at https://spraakbanken.gu.se/sparv/.

3.2.1 Requirements

- Node v0.10.x, which should be installed through you package manager. This should by default include NPM, the node package manager.
- Grunt[50], install using `npm install -g grunt-cli`.
- CoffeeScript, install using `npm install -g coffeescript`.
- Sass, install using `npm install sass`.

3.2.2 Frontend Configuration

The `app/config.js` file contains the configuration of the backend address, the address to the default settings JSON schema and also the address to Karp[51].

3.2.3 Running the Frontend

For running the frontend locally (while developing) run `grunt serve`. In you browser, open `http://localhost:9010` to launch Sparv. While running `grunt serve` the CoffeeScript and Sass files are automatically compiled upon edit, additionally causing the browser window to be reloaded to reflect the new changes.

Before releasing a new version, the scripts are compiled by running `grunt` in the frontend directory. This will create a directory called `dist` which contains all the files necessary to run the frontend.

[47] https://spraakbanken.gu.se/pub/sparv.dist/sparv_frontend

[48] https://opensource.org/licenses/MIT

[49] http://httpd.apache.org/download.cgi

[50] http://gruntjs.com/

[51] https://spraakbanken.gu.se/karp/

4 The Sparv Backend

4.1 Setting up the Sparv Backend

This section describes how to get the Sparv backend up and running on your own machine. Begin by downloading the latest version from the GitHub repository[52].

4.1.1 Info

These instructions are for installation on a UNIX-like environment. For more information on the different configuration variables check the developer's guide[53].

The Sparv backend is configured to be run in combination with the catapult which is also available on GitHub (see below).

4.1.2 Requirements

- The Sparv corpus pipeline (see here[54] for installation instructions)
- The Sparv catapult[55]
- Python 3.4[56] or newer
- A WSGI server (optional but recommended)
- GCC[57] for compiling the `catapult` C extension

4.1.3 Installation

- Set up a Python virtual environment for the backend and install the requirements from `backend/html/app/requirements.txt`.
- Set the backend configuration variables in `backend/html/app/config.py`.
- When running the backend with gunicorn (recommended) set the gunicorn configuration in `backend/html/app/gunicorn_config.py`.
- Set up the catapult, for instance in `backend/data/catapult`.
- Set up a Python virtual environment for the catapult and install the requirements from `catapult/requirements.txt`.
- Set the catapult configuration variables in `catapult/config.sh`.
- From within the `catapult` directory run `make` to build `catalaunch`.
- Set up the Sparv pipeline, for instance in `backend/data/pipeline` and build the pipeline models. You will not need to setup a virtual environment for the pipeline. Set the `VENV_PATH` to the catapult virtual environment.
- Start the catapult by running `./start-server.sh` (from within the `catapult` directory).
- Run the script `index.py` in the `backend` directory. This can be done by running it directly from the python interpreter or by starting a WSGI server using gunicorn (recommended): `html/app/venv/bin/gunicorn -c html/app/gunicorn_config.py index`.
- Set up the cron jobs listed in `catapult/cronjobs` for the automatic maintenance of Sparv.

4.2 Developing the Sparv Backend

This section describes the internal structure of the Sparv backend and its usage of the catapult and can be used as a developer's guide.

[52] https://github.com/spraakbanken/sparv-backend
[53] https://spraakbanken.gu.se/en/tools/sparv/pipeline/technical-documentation
[54] https://spraakbanken.gu.se/en/tools/sparv/pipeline/installation
[55] https://github.com/spraakbanken/sparv-catapult
[56] http://python.org/
[57] http://gcc.gnu.org/install

4.2.1 BACKEND

The backend is run through the WSGI script `index.py`. It is available at https://spraakbanken.gu.se/ws/sparv.

4.2.1.1 REQUIREMENTS

- the Sparv corpus pipeline, see here[58] for installation instructions
- Python 3.4[59] or newer

4.2.1.2 PYTHON VIRTUAL ENVIRONMENT

Though it is not required, we recommend that you use a Python virtual environment to run the Sparv backend. This is the easiest way to ensure that you have all the Python dependencies needed to run the modules.

Set up a Python virtual environment as a subdirectory to the `backend/html/app` directory:

```
python3 -m venv venv
```

Activate the virtual environment and install the required Python packages:

```
source venv/bin/activate
pip install -r requirements.txt
```

You can then deactivate the virtual environment:

```
deactivate
```

4.2.1.3 BACKEND CONFIGURATION

The configuration variables are stored in `html/app/config.py`:

- `backend`: the address where the backend is hosted
- `sparv_python`: the Python path to the sparv pipeline python directory
- `sparv_backend`: the path to the backend directory
- `builds_dir`: the directory that hosts running and completed builds
- `log_dir`: location of the log files. Can be set to `None` to log to stdout.
- `sparv_models`: location of the sparv pipeline models
- `sparv_makefiles`: location of the sparv pipeline makefiles
- `secret_key`: a string of your choosing. It is needed for queries that may cause deletions of builds.
- `venv_path`: path to the activation script of the Python virtual environment (may be set to None)
- `processes`: number of processes that `make` will run while annotating
- `fileupload_ext`: extension used for builds that contain file uploads
- `socket_file`: path to the socket file used to communicate with the catapult
- `catalaunch_binary`: path to the catalaunch binary file
- `python_interpreter`: “Python” interpreter, replaced with catalaunch

When running the backend with gunicorn (recommended) you may also want to modify the configuration in `html/app/gunicorn_config.py`.

The suggested location for the Sparv pipeline is the directory `data/pipeline`.

[58] /en/tools/sparv/pipeline/installation
[59] https://www.python.org/

4.2.1.4 Running the Backend

The backend is set up to be run with gunicorn which is installed automatically inside the Python virtual environment. From the backend directory you can run the following command:

```
html/app/venv/bin/gunicorn -c html/app/gunicorn_config.py index
```

This will start a WSGI server and bind it to the socket defined in `gunicorn_config.py`. Log messages are written to the file specified in the same file (or to the terminal if nothing is specified).

The backend can also be started by running `index.py` with the Python interpreter but this is mostly used for development or debugging.

4.2.1.5 Makefile and Settings JSON Schema

The makefile for each corpus is created from a JSON object that is created by the script `html/app/schema_generator.py`. The frontend builds its form based on the requested schema. New entries can be added and the frontend should render them automatically. The file that creates the makefile is `html/app/make_makefile.py`.

4.2.2 Catapult

The catapult runs a Python instance that shares the loaded lexicons, keeps malt processes running and lowers the Python interpreter startup time. Scripts are run on the catapult with the tiny c program `catalaunch`.

4.2.2.1 Requirements

- GCC[60] for compiling the `catapult` C extension
- Python 3.4[61] or newer

4.2.2.2 Catapult setup

The catapult can for example be placed inside the `data` directory.

Just as for the Sparv backend we recomment that you use a Python virtual environment for running the catapult. Check the backend requirements for instructions. The standard location for installing the virtual environment for the catapult is inside the `catapult` directory.

After setting up the Python virtual environment you need to adapt the variable `VENV_PATH` in the Sparv pipeline in `/makefiles/Makefile.config` so it points to the catapult virtual environment.

4.2.2.3 Catapult Configuration

The configuration variables are stored in `config.sh`:

- `SPARV_PYTHON`: the path to the `sb` pipeline Python directory
- `SPARV_MODELS`: location of the `sb` pipeline models
- `SPARV_BIN`: location of the `sb` pipeline binaries
- `SPARV_MAKEFILES`: location of the pipeline makefiles
- `CATAPULT_DIR`: location of the catapult directory
- `BUILDS_DIR`: the directory that hosts running and completed builds
- `LOGDIR`: the path to the log file directory
- `CATAPULT_VENV`: the path to the Python virtual environment used by the catapult

[60] http://gcc.gnu.org/install

[61] https://www.python.org/

4.2.2.4 Running the Catapult

- Run `make` to build `catalaunch`.
- Run `./start-server.sh` to start the catapult.
- Set up the cron jobs listed in `catapult/cronjobs`. for the automatic maintenance of Sparv.

4.2.2.5 Cron jobs

The following cron jobs are used in Sparv:

- *Cleanup*: Builds that have not been accessed for 7 days are removed every midnight by issuing `https://ws.spraakbanken.gu.se/ws/sparv/cleanup?secret_key=SECRETKEY`.
- *Keep-alive*: The script `catapult/keep-alive.sh` is run every five minutes and restarts the catapult with `catapult/start-server.sh` if it does not respond to ping. Instead of setting up this cron job you can run the catapult using a process control system like supervisord.
- *Update-saldo*: The Saldo lexicon is updated daily with the script `catapult/update-saldo.sh`. This takes some time, and is therefore run during the night. The catapult is restarted afterwards by the `keep-alive.sh` script.

5 Interaction between the Sparv Components

The following image illustrates how the components involved in the Sparv web interface interact with each other and with the user, both before and during the analysis.

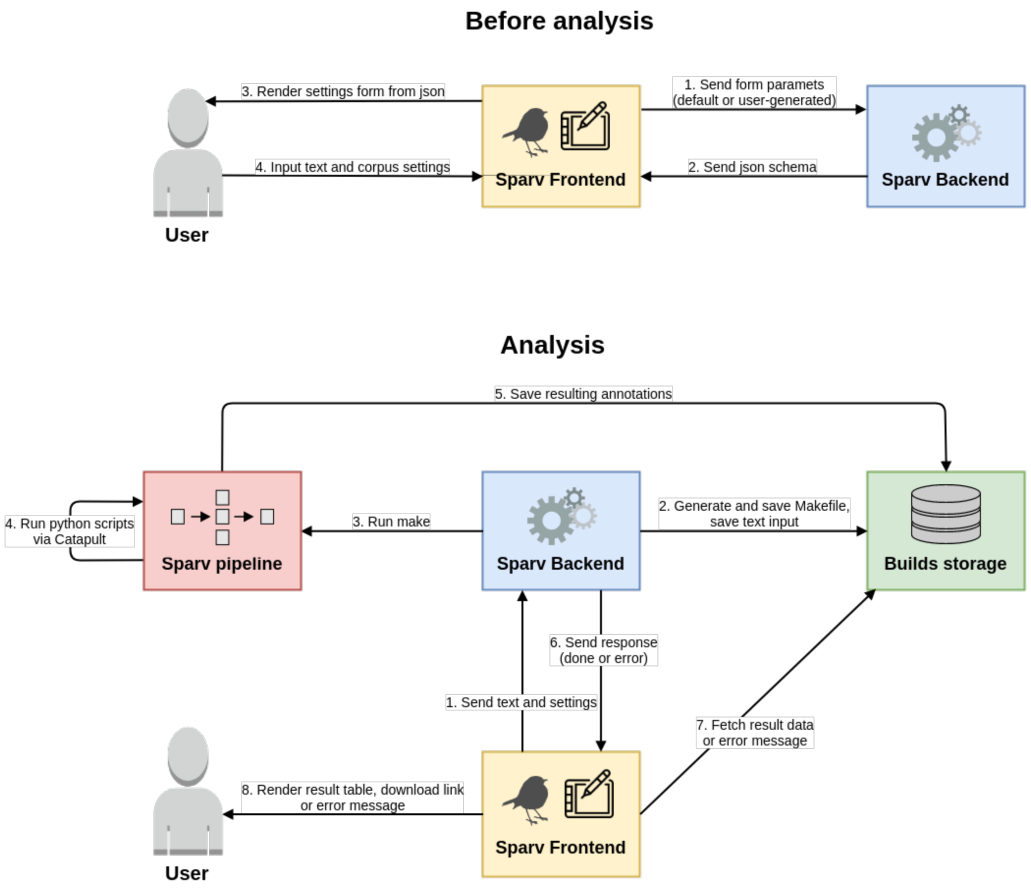
Before analysis
3. Render settings form from json
4. Input text and corpus settings
User
Sparv Frontend
1. Send form paramets (default or user-generated)
2. Send json schema
Sparv Backend
Analysis
5. Save resulting annotations
4. Run python scripts via Catapult
Sparv pipeline
3. Run make
Sparv Backend
2. Generate and save Makefile, save text input
Builds storage
6. Send response (done or error)
1. Send text and settings
7. Fetch result data or error message
User
8. Render result table, download link or error message
Sparv Frontend

Figure 1: Interaction between the Sparv components