INSTITUTIONEN FÖR SVENSKA,
FLERSPRÅKIGHET OCH SPRÅKTEKNOLOGI

GÖTEBORGS
UNIVERSITET

GU-ISS-2022-02

Sparv 5
User Manual

Martin Hammarstedt, Anne Schumacher, Lars Borin, Markus Forsberg

Forskningsrapporter från Institutionen för svenska, flerspråkighet
och språkteknologi, Göteborgs universitet
Research Reports from the Department of Swedish, Multilingualism, Language Technology

ISSN 1401-5919

Contents

1 Quick Start

This quick start guide will get you started with Sparv in just a few minutes, and will guide you through annotating your first corpus. For a more comprehensive installation and user guide, please refer to the full documentation.

> **Info:** Sparv is a command line application, and just like the 2004 Steven Spielberg movie, this quick start guide takes place in a terminal.
> This guide should work both in a Unix-like environment and the Windows command line.

1.1 Installation

Begin by making sure that you have Python 3.6.2 or newer installed by running the following in your terminal:

```
python3 --version
```

> **Note:** On some systems, the command may be called `python` instead of `python3`.

Continue by installing pipx if you haven't already:

```
python3 -m pip install --user pipx
python3 -m pipx ensurepath
```

Once pipx is installed, run the following command to install the Sparv Pipeline:

```
pipx install sparv-pipeline
```

To verify that the installation was successful, try running Sparv which should print Sparv's command line help:

```
sparv
```

Finish the installation by running the Sparv setup command, to select a location for Sparv to save its models and configuration:

```
sparv setup
```

1.2 Creating a Corpus

Now that Sparv is installed and working, let's try it out on a small corpus.

Each corpus needs its own directory, so begin by creating one called `mycorpus`:

```
mkdir mycorpus
cd mycorpus
```

In this directory, create another directory called `source`, where we will put the corpus source files (the files containing the text we want to annotate):

```
mkdir source
```

Next, use your favourite plain text editor (i.e. not Word) to create a source file in XML format, and put it in the `source` directory. Make sure to save it in UTF-8 encoding.

`document.xml`

```
<text title="My first corpus document" author="me">
    Ord, ord, ord. Här kommer några fler ord.
</text>
```

> **Note:** The `source` directory may contain as many files as you want, but let's start with just this one.

1.3 Creating the Config File

For Sparv to know what to do with your corpus, you first need to create a configuration file. This can be accomplished either by running the corpus config wizard, or by writing it by hand. Using the wizard is usually easier, but for now, let's get our hands dirty and write it by hand!

Use your text editor to create a file called `config.yaml` directly under your corpus directory. Remember to save it in UTF-8 encoding. The directory structure should now look like this:

```
mycorpus/
├── config.yaml
└── source/
    └── document.xml
```

Add the following to the configuration file and save it:

```
metadata:
    language: swe
import:
    importer: xml_import:parse
export:
    annotations:
        - <sentence>
        - <token>
```

The configuration file consists of different sections, each containing configuration variables and their values. First, we have told Sparv the language of our corpus (Swedish). Second, in the `import` section, we have specified which of Sparv's importer modules to use (we want the one for XML). Finally, in the `export` section, we have listed what automatic annotations we want Sparv to add. For this simple corpus we only ask for sentence segmentation and tokenisation.

1.4 Running Sparv

If you have followed the above steps, everything should now be ready. Make sure that you are in the `mycorpus` folder, and then run Sparv by typing:

```
sparv run
```

After a short while, Sparv will tell you where the resulting files are saved. Let's have a look at one of them:

`export/xml_export.pretty/document_export.xml`

```xml
<?xml version='1.0' encoding='UTF-8'?>
<text author="me" title="My first corpus document">
  <sentence>
    <token>Ord</token>
    <token>,</token>
    <token>ord</token>
    <token>,</token>
    <token>ord</token>
    <token>.</token>
  </sentence>
  <sentence>
    <token>Här</token>
    <token>kommer</token>
    <token>några</token>
    <token>fler</token>
    <token>ord</token>
    <token>.</token>
  </sentence>
</text>
```

1.5 What's Next?

Try adding some more annotations to your corpus by extending the annotations list in the corpus configuration. To find out what annotations are available, use the `sparv modules` command. You can also try out the corpus configuration wizard by running `sparv wizard`.

It is also possible to annotate texts in other languages, e.g., English. Just change the line `language: swe` to `language: eng` in the file `config.yaml`. Run `sparv languages` to see what languages are available in Sparv.

> **Note:** Some annotations may require additional software to be installed before you can use them.

2 Installation and Setup

This section describes how to get the Sparv Pipeline up and running on your own machine. It also describes additional software that you may need to install in order to run all the analyses provided through Sparv.

2.1 Prerequisites

In order to install Sparv you will need a Unix-like environment (e.g. Linux, macOS or Windows Subsystem for Linux) with Python 3.6.2 or newer.

> **Note:** Most of Sparv's features should work in a Windows environment as well, but since we don't do any testing on Windows we cannot guarantee anything.

2.2 Installing Sparv

Sparv is available on PyPI and can be installed using pip or pipx. We recommend using pipx, which will install Sparv in an isolated environment while still making it available to be run from anywhere.

Begin by installing pipx if you haven't already:

```
python3 -m pip install --user pipx
python3 -m pipx ensurepath
```

Once pipx is installed, run the following command to install the Sparv Pipeline:

```
pipx install sparv-pipeline
```

To verify that your installation of Sparv was successful, run the command `sparv`. The Sparv help should now be displayed.

> **Note:** After upgrading your system's Python version pipx may sometimes stop working. Some issues can be resolved by running `pipx reinstall-all`. If this fails, try to manually remove the pipx local environment directory (by default located in `~/.local/pipx`) and reinstall Sparv.

2.3 Setting Up Sparv

2.3.1 Sparv Data Directory

Sparv needs access to a directory on your system where it can store data, such as language models and configuration files. This is called the **Sparv data directory**. By running `sparv setup` you can tell Sparv where to set up its data directory. This will also populate the data directory with default configurations and presets.

> **Tip:** Instead of setting the data directory path using `sparv setup`, you may use the environment variable `SPARV_DATADIR`. This will ignore any path you have previously configured using the setup process. Note that you still have to run the setup command at least once to populate the directory, even when using the environment variable.

2.3.2 Optional: Pre-build Models

If you like, you can pre-build the model files. This step is optional, and the only advantage is that annotating your first corpus will be quicker since all the models are already set up. If you skip this step, models will be downloaded and built automatically on demand when annotating your first corpus. Pre-building models can be done by using the command `sparv build-models`. If you do this in a directory where there is no corpus config you have to tell Sparv what language the models should be built for (otherwise the language of the corpus config is used). The language is provided as a three-letter code with the `--language` flag (use the `sparv languages` command for a list of available languages and their codes). For example, if you would like to build all the Swedish models you can run `sparv build-models --language swe`.

2.4 Installing Additional Third-party Software

The Sparv Pipeline can be used together with several plugins and third-party software. Installation of the software listed below is optional. Which of these you will need to install depends on what analyses you want to run with Sparv. Please note that different licenses may apply for different software.

Unless stated otherwise in the instructions below, you won't have to download any additional language models or parameter files. If the software is installed correctly, Sparv will download and install the necessary model files for you prior to annotating data.

2.4.1 Sparv wsd

Purpose	Swedish word-sense disambiguation. Recommended for standard Swedish annotations.
Download	Sparv wsd
License	MIT
Dependencies	Java

Sparv wsd is developed at Språkbanken and runs under the same license as the Sparv Pipeline. In order to use it within the Sparv Pipeline it is enough to download the saldowsd.jar from GitHub (see download link above) and place it inside your Sparv data directory under `bin/wsd`.

2.4.2 hfst-SweNER

Purpose	Swedish named-entity recognition. Recommended for standard Swedish annotations.
Download	hfst-SweNER
Version compatible with Sparv	0.9.3

Note: hfst-SweNER requires a Unix-like environment.

The current version of hfst-SweNER is written for Python 2 while Sparv uses Python 3, so before installing it needs to be patched. After extracting the archive, go to the `hfst-swener-0.9.3/scripts` directory and create the file `swener.patch` with the following contents:

```
--- convert-namex-tags.py
+++ convert-namex-tags.py
@@ -1 +1 @@
-#! /usr/bin/env python
+#! /usr/bin/env python3
@@ -34 +34 @@
-         elif isinstance(files, basestring):
+         elif isinstance(files, str):
@@ -73 +73 @@
-         return [s[start:start+partlen] for start in xrange(0, len(s), partlen)]
+         return [s[start:start+partlen] for start in range(0, len(s), partlen)]
@@ -132,3 +131,0 @@
-     sys.stdin = codecs.getreader('utf-8')(sys.stdin)
-     sys.stdout = codecs.getwriter('utf-8')(sys.stdout)
-     sys.stderr = codecs.getwriter('utf-8')(sys.stderr)
```

Then simply run the command `patch < swener.patch`, which will make the necessary changes.

After applying the patch, please follow the installation instructions provided by hfst-SweNER.

2.4.3 Hunpos

Purpose	Alternative Swedish part-of-speech tagger (if you don't want to use Stanza)
Download	Hunpos on Google Code
License	BSD-3
Version compatible with Sparv	latest (1.0)

Installation is done by unpacking and then adding the executables to your path (you will need at least `hunpos-tag`). Alternatively you can place the binaries inside your Sparv data directory under `bin`.

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 lib32z1
```

On newer macOS you probably have to compile Hunpos from source. This GitHub repo has instructions that should work.

When using Sparv with Hunpos on Windows you will have to set the config variable `hunpos.binary : hunpos-tag.exe` in your corpus configuration. You will also have to add the `cygwin1.dll` file that comes with Hunpos to your path or copy it into your Sparv data directory along with the Hunpos binaries.

2.4.4 MaltParser

Purpose	Alternative Swedish dependency parser (if you don't want to use Stanza)
Download	MaltParser webpage
License	MaltParser license (open source)
Version compatible with Sparv	1.7.2
Dependencies	Java

Download and unpack the zip-file from the MaltParser webpage and place the `maltparser-1.7.2` directory inside the `bin` directory of the Sparv data directory.

2.4.5 Corpus Workbench

Purpose	Creating Corpus Workbench binary files. Only needed if you want to be able to search corpora with this tool.
Download	Corpus Workbench on SourceForge
License	GPL-3.0
Version compatible with Sparv	beta 3.4.21 (most likely works with newer versions)

Refer to the INSTALL text file for instructions on how to build and install on your system.

2.4.6 Software for Analysing Other Languages than Swedish

Sparv can use different third-party tools for analyzing corpora in other languages than Swedish.

The following is a list of the languages currently supported by Sparv, their language codes (ISO 639-3) and which tools Sparv can use to analyse them:

Language	ISO 639-3 Code	Analysis Tool
Asturian	ast	FreeLing
Bulgarian	bul	TreeTagger
Catalan	cat	FreeLing
Dutch	nld	TreeTagger
Estonian	est	TreeTagger
English	eng	FreeLing, Stanford Parser, TreeTagger
French	fra	FreeLing, TreeTagger
Finnish	fin	TreeTagger
Galician	glg	FreeLing
German	deu	FreeLing, TreeTagger
Italian	ita	FreeLing, TreeTagger
Latin	lat	TreeTagger
Norwegian Bokmål	nob	FreeLing
Polish	pol	TreeTagger
Portuguese	por	FreeLing
Romanian	ron	TreeTagger
Russian	rus	FreeLing, TreeTagger
Slovak	slk	TreeTagger
Slovenian	slv	FreeLing
Spanish	spa	FreeLing, TreeTagger
Swedish	swe	Sparv

2.4.6.1 TreeTagger

Purpose	POS-tagging and lemmatisation for some languages
Download	TreeTagger webpage
License	TreeTagger license (freely available for research, education and evaluation)
Version compatible with Sparv	3.2.3 (may work with newer versions)

After downloading the software you need to have the `tree-tagger` binary in your path. Alternatively you can place the `tree-tagger` binary file in the Sparv data directory under `bin`.

2.4.6.2 Stanford Parser

Purpose	Various analyses for English
Download	Stanford CoreNLP webpage
License	GPL-2.0
Version compatible with Sparv	4.0.0 (may work with newer versions)
Dependencies	Java

Please download, unzip and place contents inside the Sparv data directory under `bin/stanford_parser`.

2.4.6.3 FreeLing

Purpose	Tokenisation, POS-tagging, lemmatisation and named entity recognition for some languages
Download	FreeLing on GitHub
License	AGPL-3.0
Version compatible with Sparv	4.2

Please install the software (including the additional language data) according to the instructions provided by FreeLing. Note that you will need to uncompress the source and language files in the same folder before compiling. You will also need to install the sparv-sbx-freeling plugin. Please follow the installation instructions for the sparv-sbx-freeling module on GitHub in order to set up the plugin correctly.

2.5 Plugins

If you have the Sparv Pipeline installed on your machine, you can install plugins by injecting them into the Sparv Pipeline code using pipx:

```
pipx inject sparv-pipeline [pointer-to-sparv-plugin]
```

The `pointer-to-sparv-plugin` can be a package available on the Python Package Index (PyPI), a remote public repository, or a local directory on your machine.

For now there are two plugins available for Sparv: sparv-sbx-freeling and sparv-sbx-metadata. Please refer to their GitHub page for more information.

Plugins can be uninstalled by running:

```
pipx runpip sparv-pipeline uninstall [name-of-sparv-plugin]
```

2.6 Uninstalling Sparv

To uninstall Sparv completely, follow these steps:

1. Run `sparv setup --reset` to unset Sparv's data directory. The directory itself will not be removed, but its location (if available) will be printed.
2. Manually delete the data directory.

3. Run one of the following commands, depending on whether you installed Sparv using pipx or pip:

   ```
   pipx uninstall sparv-pipeline
   ```

   ```
   pip uninstall sparv-pipeline
   ```

3 Running Sparv

Sparv is run from the command line. Typically, you will want to run Sparv from within a corpus directory containing some text files (the corpus) and a corpus config file. A typical corpus directory structure could look like this:

```
mycorpus/
├── config.yaml
└── source
    ├── document1.xml
    ├── document2.xml
    └── document3.xml
```

When trying out Sparv for the first time we recommend that you download and test run some of the example corpora.

When running `sparv` (or `sparv -h`) the available sparv commands will be listed:

```
Annotating a corpus:
    run              Annotate a corpus and generate export files
    install          Annotate and install a corpus on remote server
    clean            Remove output directories

Inspecting corpus details:
    config           Display the corpus config
    files            List available corpus source files (input for Sparv)

Show annotation info:
    modules          List available modules and annotations
    presets          List available annotation presets
    classes          List available annotation classes
    languages        List supported languages

Setting up the Sparv Pipeline:
    setup            Set up the Sparv data directory
    wizard           Run config wizard to create a corpus config
    build-models     Download and build the Sparv models (optional)

Advanced commands:
    run-rule         Run specified rule(s) for creating annotations
    create-file      Create specified file(s)
    run-module       Run annotator module independently
    preload          Preload annotators and models
```

Every command in the Sparv command line interface has a help text which can be accessed with the `-h` flag. Below we will give an overview of the most important commands in Sparv.

3.1 Annotating a Corpus

`sparv run`: From inside a corpus directory with a config file you can annotate the corpus using `sparv run`. This will start the annotation process and produce all the output formats (or *exports*) listed under `export.default` in your config. You can also tell Sparv explicitly what output format to generate, e.g. `sparv run csv_export:csv`. Type `sparv run -l` to learn what output formats there are available for your corpus. The output files will be stored in a directory called `export` inside your corpus directory.

`sparv install`: Installing a corpus means deploying it in some way, either locally or on a remote server. Sparv supports deployment of compressed XML exports, CWB data files and SQL data. If you try to install a corpus, Sparv will check if the necessary annotations have been created. If any annotations are missing, Sparv will run them for you. Therefore, you do not need to annotate the corpus before installing. You can list the available installation options with `sparv install -l`.

`sparv clean`: While annotating, Sparv will create a directory called `sparv-workdir` inside your corpus directory. You normally don't need to touch the files stored here. Leaving this directory as it is will usually lead to faster processing of your corpus if you for example want to add a new output format. However, if you would like to delete this directory (e.g. because you want to save disk space or because you want to rerun all annotations from scratch) you can do so by running `sparv clean`. The export directory and log files can also be removed with the `clean` command by adding appropriate flags. Check out the available options (`sparv clean -h`) to learn more.

3.2 Show Annotation Info

`sparv modules`: List available modules and annotations.

`sparv presets`: List available annotation presets available for your corpus. You can read more about presets in the section about annotation presets.

`sparv classes`: List available annotation classes. You can read more about classes in the section about annotation classes.

`sparv languages`: List supported languages.

3.3 Inspecting Corpus Details

`sparv config`: This command lets you inspect the configuration for your corpus. You can read more about this in the section about corpus configuration.

`sparv files`: By using this command you can list all available source files belonging to your corpus.

3.4 Setting Up the Sparv Pipeline

`sparv setup` and **`sparv build-models`:** These commands are explained in the section Setting Up Sparv.

3.5 Advanced Commands

`sparv run-rule` and **`sparv create-file`:** Instruct Sparv to run the specified annotation rule or to create the specified file. Multiple arguments can be supplied.

Example running the Stanza annotations (part-of-speech tagging and dependency parsing) for all input files:

```
sparv run-rule stanza:annotate
```

Example creating the part-of-speech annotation for the input file `document1`:

```
sparv create-file annotations/dokument1/segment.token/stanza.pos
```

`sparv run-module`: Run an annotator module independently (mostly for debugging). You must supply the module and the function you want to run and all the mandatory arguments. E.g. to run the hunpos msd tagging module on the input file called `document1` you could use the following command:

```
sparv run-module hunpos msdtag --out segment.token:hunpos.msd --word segment.
    token:misc.word --sentence segment.sentence --binary hunpos-tag --model
    hunpos/suc3_suc-tags_default-setting_utf8.model --morphtable hunpos/saldo_suc
    -tags.morphtable --patterns hunpos/suc.patterns --encoding UTF-8 --
    source_file dokument1
```

`sparv preload`: This command preloads annotators and their models and/or related binaries to speed up annotation. This is especially useful when annotating multiple smaller source files, where every model otherwise would have to be loaded as many times as there are source files. Not every annotator supports preloading; use the `--list` argument to see which annotators are supported.

The Sparv preloader can be run from anywhere as long as there is a `config.yaml` file in the same directory. While the file follows the same format as all corpus configuration files, it doesn't necessarily have to be tied to a corpus. All that is required is a `preload:` section, with a list of annotators to preload (from the list given by the command above). The listed annotators will be loaded using the settings in the configuration file (in combination with default settings, as usual).

The Sparv preloader may be shared between several corpora, as long as the configuration for the annotators doesn't differ (e.g. what models are used). Sparv will automatically fall back to not using the preloader for a certain annotator if it detects that the preloaded version is using a different configuration from what is needed for the corpus.

The preloader uses socket files for communication. Use the `--socket` argument to provide a path to the socket file to create. If omitted, the default `sparv.socket` will be used.

The `--processes` argument tells Sparv how many parallel processes to start. If possible, use as many processes as you plan on using when running Sparv (e.g. `sparv run -j 4`), or the preloader might become a bottleneck instead of speeding things up.

Example of starting the preloader with four parallel processes:

```
sparv preload --socket my_socket.sock --processes 4
```

Once the preloader is up and running, use another terminal to annotate your corpus. To make Sparv use the preloader when annotating, use the `--socket` argument and point it to the same socket file created by the preloader. For example:

```
sparv run --socket my_socket.sock
```

If the preloader is busy, by default Sparv will execute annotators the regular way without using the preloader. If you would rather have Sparv wait for the preloader, use the `--force-preloader` flag with the `run` command.

To shut down the preloader, either press Ctrl-C in the preloader terminal, or use the command `sparv preload stop` while pointing it to the relevant socket. For example:

```
sparv preload stop --socket my_socket.sock
```

4 Requirements for Source Files

In order for Sparv to be able to process your corpus, please make sure your source files meet the following requirements:

1. Make sure you don't have any manually created directories called `sparv-workdir` or `export` in your corpus directory as Sparv will attempt to create and use these.
2. If your corpus is in XML format, make sure your **XML is valid** and that the text to be analysed is actual text (not attribute values).
3. Your source files must all use the same file format, same file extension and (if applicable) the same markup.
4. If your source files are very large or if your corpus consists of a large number of tiny files, Sparv may become quite slow. Very large files may also lead to memory problems. Try keeping the maximum file size per file around 5-10 MB, and in the case of many tiny files, combining them into larger files if possible. If your machine has a lot of memory, processing larger files may work just fine.

5 Corpus Configuration

To be able to annotate a corpus with Sparv you will need to create a corpus config file. A corpus config file is written in YAML, a fairly human-readable format for creating structured data. This file contains information about your corpus (metadata) and instructions for Sparv on how to process it. The corpus config wizard can help you create one. If you want to see some examples of config files you can download the example corpora.

A minimal config file contains a list of (automatic) annotations you want to be included in the output. Here is an example of a small config file:

```
metadata:
    # Language of the source files
    language: swe
export:
    # Automatic annotations to be included in the export
    annotations:
        - <sentence>:misc.id
        - <token>:saldo.baseform
        - <token>:hunpos.pos
        - <token>:sensaldo.sentiment_label
```

> **Note:** In Sparv and this documentation, configuration keys are often referred to using dot notation, like this: `export.annotations`. This should be interpreted as the configuration key `annotations` nested under the section `export`, as shown in the example above.

> **Note:** Most annotators in Sparv have one or more options that can be fine-tuned using the configuration file. Each module has its own section in the file, like the `metadata` and `export` sections in the example above. By using the `sparv modules` command you can get a list of the available configuration keys and their descriptions.

5.1 Corpus Config Wizard

The corpus config wizard is a tool designed to help you create a corpus config file by asking questions about your corpus and the annotations you would like Sparv to add to it. Run `sparv wizard` in order to start the tool. When running this command in a directory where a corpus config file exists already, Sparv will read the config file and set the wizard default values according to the existing configuration.

The wizard is an auxiliary tool to get you started with your corpus config file, and it does not cover all of Sparv's advanced functionality. However, a config file that was created with the wizard can of course be edited manually afterwards, e.g. for adding more advanced configuration details such as custom annotations or headers.

5.2 Default Values

Some config variables such as `metadata`, `classes`, `import`, `export` and `custom_annotations` are general and are used by multiple Sparv modules, while others are specific to one particular annotation module (e.g. `hunpos.binary` defines the name of the binary the hunpos module uses to run part-of-speech tagging). These module specific config options usually have default values which are defined by the module itself.

When running Sparv your corpus config will be read and combined with Sparv's default config file (`config_default.yaml` in the Sparv data directory) and the default values defined by different Sparv modules. You can view the resulting configuration by running `sparv config`. Using the `config` command you can also inspect specific config variables, e.g. `sparv config metadata` or `sparv config metadata.language`. All default values can be overridden in your own corpus config.

There are a few config options that must be set (either through the default config or the corpus config): - `metadata.language` (default: `swe`) - `import.importer` (default: `xml_import:parse`) - `export.annotations` - `classes.token` (default: `segment.token`) - `classes.sentence` (default: `segment.sentence`)

5.3 Metadata Options

The `metadata` section of your corpus config contains metadata about your corpus that may be used by any Sparv module.

- `metadata.id` defines the machine name of the corpus. It is required by some exporter modules. This string may contain ascii letters, digits and dashes.
- `metadata.name` is an optional human readable name of the corpus. This option is split into two fields, `eng` and `swe` for defining a name in English and in Swedish.
- `metadata.language` defines the language of the source files in the corpus. This should be an ISO 639-3 code. If not specified it defaults to `swe`. Run `sparv languages` to list the supported languages along with their language codes.
- `metadata.variety` is an optional field containing the language variety of the source files (if applicable). Run `sparv languages` to list the supported varieties for each language.
- `metadata.description` is an optional description for the corpus. It may consist of multiple lines. This option is split into two fields, `eng` and `swe` for defining a name in English and in Swedish.

5.4 Import Options

The `import` section of your corpus config is used to give Sparv some information about your input files (i.e. your corpus).

- `import.source_dir` defines the location of your input files and it defaults to `source`. Sparv will check the source directory recursively for valid input files to process.
- `import.importer` is used to tell Sparv which importer to use when processing your source files. The setting you want to choose depends on the format of your input files. If your corpus is in XML you should choose `xml_import:parse` (this is the default setting). If your corpus files are in plain text, you should choose `text_import:parse` instead.
- `import.text_annotation` specifies the annotation representing *one text*, and any automatic text-level annotations will be attached to this annotation. For XML source files this refers to one of the XML elements. For plain text source files a default `text` root annotation will be created automatically, and you won't have to change this setting.

  > **Note:** This setting automatically sets the `text` class. If you want to use an automatic annotation as the text annotation, you should not use this setting, and instead set the `text` class directly.

- `import.encoding` specifies the encoding of the source files. It defaults to UTF-8.
- `import.normalize` lets you normalize unicode symbols in the input using any of the following forms: 'NFC', 'NFKC', 'NFD', and 'NFKD'. It defaults to `NFC`.
- `import.keep_control_chars` may be set to `True` if control characters should not be removed from the text. You normally don't want to enable this, since it most likely will lead to problems.

Each importer may have additional options which can be listed with `sparv modules --importers`. The XML importer for example has options that allow you to skip importing the contents of certain elements and options that give you fine-grained control over importing XML headers. Run `sparv modules --importers xml_import` for more details.

5.5 Export Options

The `export` section of your corpus config defines what the output data (or export) should look like. With the config option `export.source_annotations` you can tell Sparv what elements and attributes present in your source files you would like to keep in your output data (this only applies if your input data is XML). If you don't specify anything, everything will be kept in the output. If you do not want any source annotations to be included in your output you can set this option to `[]`. This will cause errors in the XML exports though because the root element must be listed as a source annotation. If you do list anything here, make sure that you include the root element (i.e. the element that encloses all other included elements and text content) for each of your input files. If you don't, the resulting output XML will be invalid and Sparv won't be able to produce XML files. If you only want to produce other output formats than XML, you don't need to worry about this.

It is possible to rename elements and attributes present in your input data. Let's say your files contain elements like this `<article name="Scandinavian Furniture" date="2020-09-28">` and you would like them to look like this in the output `<text title="Scandinavian Furniture" date="2020-09-28">` (so you want to rename the element "article" and the attribute "name" to "text" and "title" respectively). For this you can use the following syntax:

```
export:
    source_annotations:
        - article as text
        - article:name as title
        - ...
```

Please note that the dots (`...`) in the above example also carry meaning. They are used to refer to all the remaining elements and attributes in your input data. Without using the dots the "date" attribute in the example would be lost.

If you want to keep most of the markup of your input data but you want to exclude some elements or attributes you can do this by using the `not` keyword:

```
export:
    source_annotations:
        - not date
```

In the example above this would result in the following output: `<article name="Scandinavian Furniture">`.

The option `export.annotations` contains a list of automatic annotations you want Sparv to produce and include in the output. You can run `sparv modules --annotators` to see what annotations are available. Some annotations listed here contain curly brackets, e.g. `{annotation}:misc.id`. This means that the annotation contains a wildcard (or in some cases multiple wildcards) that must be replaced with a value when using the annotation in the `export.annotations` list (e.g. `<sentence>:misc.id`). You can also read the section about annotation presets for more info about automatic annotations.

If you want to produce multiple output formats containing different annotations you can override the `export.source_annotations` and `export.annotations` options for specific exporter modules. The annotations for the XML export for example are set with `xml_export.source_annotations` and `xml_export.annotations`, the annotations for the CSV export are set with `csv_export.source_annotations` and `csv_export.annotations` and so on. Many of the `export` options work this way, where the values from `export` will be used by default unless overridden on exporter module level.

Tip: If two or more sections of your config are identical, for example the list of annotations to include in different export formats, instead of copying and pasting you can use YAML anchors.

Tip: It is possible to convert a structural attribute to a token attribute, which can be convenient for representing structural information (such as named entities or phrase structures) in non-structured formats (e.g. the CSV export). Use the annotation `<token>:misc.from_struct_{struct}_{attr}` where you replace `{struct}` and `{attr}` with the name of the structural annotation and the attribute name respectively (e.g. `<token>:misc.from_struct_swener.ne_swener.type`).

The option `export.default` defines a list of export formats that will be produced when running `sparv run` without format arguments. By default, this list only contains `xml_export:pretty`, the formatted XML export with one token per line. Use the command `sparv run --list` to see a list of available export formats.

There are a couple of export options concerning the naming of annotations and attributes. You can choose to prefix all annotations produced by Sparv with a custom prefix with the `export.sparv_namespace` option. Likewise, you can add a prefix to all annotations and attributes originating from your source files with the `export.source_namespace` option.

The option `export.remove_module_namespaces` is `true` by default, meaning that module name prefixes are removed during export. Turning the option off will result in output like:

```
<segment.token stanza.pos="IN" saldo.baseform="|hej|">Hej</segment.token>
```

instead of the more compact:

```
<token pos="IN" baseform="|hej|">Hej</token>
```

`export.scramble_on` is a setting used by all the export formats that support scrambling. It controls which annotation your corpus will be scrambled on. Typical settings are `export.scramble_on: <sentence>` or `export.scramble_on: <paragraph>`. For example, setting this to `<paragraph>` would lead to all paragraphs being randomly shuffled in the export, while the sentences and tokens within the paragraphs keep their original order.

The option `export.word` is used to define the strings to be output as tokens in the export. By default, this is set to `<token:word>`. A useful application for this setting is anonymisation of texts. If you want to produce XML containing only annotations but not the actual text, you could set `export.word: <token>:anonymised` to get output like this:

```
    <sentence id="b1ac">
      <token pos="IN">***</token>
      <token pos="MAD">*</token>
    </sentence>
```

Note: For technical reasons the export `xml_export:preserved_format` does not respect this setting. The preserved format XML will always contain the original corpus text.

Each exporter may have additional options which can be listed with `sparv modules --exporters`.

5.6 Headers

Sometimes corpus metadata in XML is stored in headers rather than in attributes belonging to text-enclosing elements. Sparv can extract information from headers and store as annotations. These can then be used as input for different analyses. Information from headers can be exported as attributes if you choose to.

Let's say we have a corpus file with the following contents:

```
<text id="1">
    <header>
        <author birth="1780" death="????">Anonym</author>
        <date>2020-09-08</date>
        <title>
            <main-title>A History of Corpora</main-title>
            <sub-title>A Masterpiece</sub-title>
        </title>
    </header>
    <another-header>
        <dummy>1</dummy>
    </another-header>
    Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do
    eiusmod tempor incididunt ut labore et dolore magna aliqua.
</text>
```

We want to keep the data in `<header>` but we don't want the contents to be analysed as corpus text. Instead, we want its metadata to be attached to the `<text>` element. We also want to get rid of `<another-header>` and its contents entirely. This configuration will do the job:

```
xml_import:
    header_elements:
        - header
        - another-header
    header_data:
        - header/author as text:author
        - header/author:birth as text:author-birth
        - header/author:death as text:author-death
        - header/title/main-title as text:title
        - header/title/sub-title as text:subtitle
        - header/date as text:date
xml_export:
    header_annotations:
        - not header
        - not another-header
```

The output will look like this:

```
<text author="Anonym" author-birth="1780" author-death="????" date="2020-09-08"
      id="1" title="A History of Corpora" subtitle="A Masterpiece">
    <sentence id="13f">
      <token>Lorem</token>
      <token>ipsum</token>
      ...
    </sentence>
</text>
```

If you do want to keep the headers in the output (without them being analysed as corpus text), just list them without the `not` prefix in `xml_export.header_annotations`. If you don't specify anything at all in `xml_export.header_annotations` all your headers will be kept.

5.7 XML Namespaces

If the source data is in XML and contains namespaces Sparv will try to keep these intact in the XML output. There are, however, two limitations: 1. Namespace declarations are always placed in the root element in the output, regardless of where they are in the source data. 2. URIs and prefixes are assumed to be unique. A URI will automatically be associated with the first prefix that is declared for that URI in the source file.

When referring to elements or attributes containing namespaces in the corpus config file a special syntax is used. A reference consists of the namespace prefix followed by +, followed by the tag or attribute name. E.g. the reference for this element `<sparv:myelement xmlns:sparv="https://spraakbanken.gu.se/verktyg/sparv">` would be `sparv+myelement`.

Namespaces may be removed upon import by setting `xml_import.remove_namespaces` to `true` in the corpus config. This may however result in collisions in attributes containing namespaces in the source data.

5.8 Annotation Classes

Annotation classes are used to refer to annotations without having to explicitly point out which module produces that annotation. Two examples we have already shown in examples above are `<token>` and `<sentence>`, which by default refer to the annotations `segment.token` and `segment.sentence` respectively.

Annotation classes are used internally to simplify dependence relations between modules, and to increase the flexibility of the pipeline. For example, a part-of-speech tagger that requires tokenised text as input probably doesn't care about which tokeniser is used, so it simply asks for `<token>`.

Annotation classes are also useful for you as a user. In most places in the config file where you refer to annotations, you can also use classes. To see what classes are available, use the command `sparv classes`. Classes are referred to by enclosing the class name with angle brackets: `<token>`.

If a class can be produced by more than one module, you can use the `classes` section in your config file to select which one to use. Sparv already comes with a few default class settings.

One example of a config section where you can use classes is the export annotations list. If you want to include part-of-speech tags from Stanza you could include `segment.token:stanza.pos` in the annotations list, meaning that you want to have POS tags from Stanza as attributes to the tokens produced by the segment module.

```
export:
    annotations:
        - segment.token:stanza.pos
        - segment.token:malt.deprel
        - segment.token:malt.dephead_ref
```

The disadvantage of this notation is that if you decide to exchange your tokeniser for a different one called `my_new_segmenter` you would have to change all occurrences of `segment.token` in your config to `my_new_segmenter.token`. Instead of doing that you can just re-define your token class by setting `classes.token` to `my_new_segmenter.token` and use the `<token>` class in your annotations list:

```
classes:
    token: my_new_segmenter.token

export:
    annotations:
        - <token>:stanza.pos
        - <token>:malt.deprel
        - <token>:malt.dephead_ref
```

Re-defining annotation classes may also be necessary when your corpus data contains annotations (such as sentences or tokens) that should be used as input to annotators. For example, if you have done manual sentence segmentation and enclosed each sentence in an `<s>` element, you can skip Sparv's automatic sentence segmentation by setting the sentence class to this element:

```
classes:
    sentence: s

xml_import:
    elements:
        - s
```

> **Attention:** Please note that you need to tell Sparv that `s` is an annotation imported from your corpus data. This is done by listing `s` under `xml_import.elements` as is done in the above example.

5.9 Annotation Presets

Annotation presets are collections of annotations which can be used instead of listing the contained annotations. For example, instead of listing all the SALDO annotations in your list of automatic annotations like this:

```
export:
    annotations:
        - <token>:saldo.baseform2 as baseform
        - <token>:saldo.lemgram
        - <token>:wsd.sense
        - <token>:saldo.compwf
        - <token>:saldo.complemgram
```

… you can use the `SWE_DEFAULT.saldo` preset:

```
export:
    annotations:
        - SWE_DEFAULT.saldo
```

Here `SWE_DEFAULT.saldo` will expand to all the SALDO annotations. You can mix presets with annotations, and you can combine different presets with each other.

Sparv comes with a set of default presets, and you can use the `sparv presets` command to see which are available for your corpus, and which annotations they include. Presets are defined in YAML files and can be found in the Sparv data directory under `config/presets`. You can also add your own presets just by adding YAML files to this directory.

It is possible to exclude specific annotations from a preset by using the `not` keyword. In the following example we are including all SALDO annotations except for the compound analysis attributes:

```
export:
    annotations:
        - SWE_DEFAULT.saldo
        - not <token>:saldo.compwf
        - not <token>:saldo.complemgram
```

> **Note:** Preset files may define their own `class` default values. These will be set automatically when using a preset. You can override these in your config files if you know what you are doing.

5.10 Parent Configuration

If you have multiple corpora with similar configurations where only some variables differ for each corpus (e.g. the corpus ID), you may add a reference to a parent configuration file from your individual corpus config files. Specify the path to the parent config file in the `parent` variable, and your corpus configuration will inherit all the parameters from it that are not explicitly specified in the individual config file. Using a list, multiple parents can be specified, each parent overriding any conflicting values from previous parents. Nested parents are also allowed, i.e. parents referencing other parents.

```
parent: ../parent-config.yaml
metadata:
    id: animals-foxes
    name:
        swe: 'Djurkorpus: Rävar'
```

The above configuration will contain everything specified inside `../parent-config.yaml` but the values for `metadata.id` and `metadata.name.swe` will be overridden with `animals-foxes` and `Djurkorpus: Rävar` respectively.

5.11 Custom Annotations

The `custom_annotations` section of the config file is used for three different purposes, each explained separately below.

5.11.1 Built-in Utility Annotations

Most annotators in Sparv can be customised by using configuration variables, but are still usable even without changing their default configuration. Sparv also comes with another type of annotator, that *needs* to be configured to be used, and instead of configuration variables, they use parameters. We call these “utility annotators”, and the purpose of these utility annotators is often to modify other annotations. The `misc:affix` annotator for example, can be used to add a prefix and/or a suffix string to another annotation.

To include a utility annotation in your corpus, it first needs to be configured in the `custom_annotations` section of your config. If we use `misc:affix` as an example, it could look like this:

```
custom_annotations:
    - annotator: misc:affix
      params:
          out: <token>:misc.word.affixed
          chunk: <token:word>
          prefix: "|"
          suffix: "|"
```

Here we are using the word annotation as input and add the string “|” as prefix and suffix. First we specify the annotator we want to use (`annotator`), and then use the `params` section to set values for the annotator’s parameters. The `sparv modules` command will show you the list of parameters for each utility annotator (and this is also how you recognise a utility annotator in that list). For this particular annotator the parameters are as follows: 1. `out`: What your output annotation should be called. The output name must always include the module name as a prefix, in this case `misc`. 2. `chunk`: What annotation you want to use as input (the chunk). 3. `prefix` and `suffix`: The string that you want to use as prefix and/or suffix.

In order to include this annotation in your corpus, you then need to add `<token>:misc.word.affixed` to an annotations list in your corpus config (e.g. `export.annotations`). This example is applied in the standard-swe example corpus.

You can use the same annotator multiple times as long as you name the outputs differently:

```
custom_annotations:
    - annotator: misc:affix
      params:
          out: <token>:misc.word.affixed
          chunk: <token:word>
          prefix: "|"
          suffix: "|"
    - annotator: misc:affix
      params:
          out: <token>:misc.word.affixed2
          chunk: <token:word>
          prefix: "+"
          suffix: "+"
```

Note: Custom annotations always result in new annotations; they do not modify existing ones.

Note: When a parameter for a custom annotator requires a regular expression (e.g. in `misc:find_replace_regex`), the expression must be surrounded by single quotation marks. Regular expressions inside double quotation marks in YAML are not parsed correctly.

5.11.2 Reusing Regular Annotations

Another use for `custom_annotations` is when you want to use a regular (non-utility) annotation more than once in your corpus. One example is if you want to use the same part-of-speech tagger multiple times, but with different models.

In order to do this you specify the annotator in the `custom_annotations` section of your corpus config, and add a `config` section, under which you add configuration for the annotator just like you would normally do in the root of the configuration file. To give this alternative annotation a unique name, you also have to add a `suffix` which will then be added to the original annotation name.

In the example below we are reusing the `hunpos:msdtag` annotator with a custom model.

```
custom_annotations:
    - annotator: hunpos:msdtag
      suffix: -mymodel
      config:
          hunpos:
              model: path/to/my_hunpos_model
```

The regular Hunpos annotation is named `<token>:hunpos.msd`, but with the specified suffix this new annotation will be named `<token>:hunpos.msd-mymodel`, which can then be referred to in the list of annotations:

```
export:
    annotations:
        - <token>:hunpos.msd
        - <token>:hunpos.msd-mymodel
```

5.11.3 User-defined Custom Annotators

Extending Sparv with new annotators is typically done by creating a plugin, which when installed becomes available to all your corpora. An alternative to creating a plugin is creating a user-defined custom annotator. It is very similar to a plugin, but is available only to the corpus in the same directory.

The full documentation for how to write a Sparv annotator can be found in the developer's guide, but here is a quick example.

> **Tip:** The following example uses the `@annotator` decorator for creating an annotator, but it is possible to create your own importer, exporter, installer or model builder using the appropriate Sparv decorator. You can read more about decorators in the developer's guide.

Creating a user-defined custom annotator involves the following three steps: 1. Create a Python script with an annotator and place it in your corpus directory 2. Register the annotator in your corpus config 3. Use your custom annotation by referring to it in an annotations list

Step 1: Add your user-defined custom annotator by creating a Python script in your corpus directory, e.g. `convert.py`:

```
mycorpus/
├── config.yaml
├── convert.py
└── source
    ├── document1.xml
    └── document2.xml
```

Sparv will automatically detect scripts placed here as long as your functions are registered in your config (see Step 2). Your annotator function must use one of the Sparv decorators (usually `@annotator`). Here is a code example for a simple annotator that converts all tokens to upper case:

```python
from sparv.api import Annotation, Output, annotator

@annotator("Convert every word to uppercase.")
def uppercase(word: Annotation = Annotation("<token:word>"),
              out: Output = Output("<token>:custom.convert.upper")):
    """Convert to uppercase."""
    out.write([val.upper() for val in word.read()])
```

Step 2: Now register your custom annotator in your corpus config in the `custom_annotations` section so Sparv can find it. The name of your annotator is composed of: - the prefix `custom.` - followed by the filename of the Python file without extension (`convert` in our example) - followed by a colon - and finally the annotator name (`uppercase`)

```
custom_annotations:
    - annotator: custom.convert:uppercase
```

Step 3: Now you can go ahead and use the annotation created by your custom annotator. Just add the annotation name given by the `out` parameter value to an annotations list in your corpus config:

```
export:
    annotations:
        - <token>:custom.convert.upper
```

In this example all parameters in the annotator function have default values which means that you do not need to supply any parameter values in your config. But of course you can override the default values:

```
custom_annotations:
    - annotator: custom.convert:uppercase
      params:
          out: <token>:custom.convert.myUppercaseAnnotation
```

> **Note:** When using custom annotations from your own code all output annotations must be prefixed with `custom`.

There is an example of a user-defined custom annotator in the standard-swe example corpus.

If you need more information on how to write an annotator function please refer to the developer's guide. If you have written a rather general annotator module, you could consider making it into a Sparv plugin. This way other people will be able to use your annotator. Read more about writing plugins in the developer's guide.