Spejd 1.3.6 - User manual

Bartosz Zaborowski

October, 2012

Contents

Chapter 1

Configuration

The main file of Spejd's configuration is called usually "config.ini". By default Spejd looks for this file in the current directory on execution. This default can be overridden by specifying `-c <path to config file>` option.

The configuration file consists of several options. Some of them are required, but most of those have reasonable default values. Below goes a list of standard options with explanations grouped by their functions.

Note 1. *If a single option is specified multiple times in the file, the last occurrence is used.*

Note 2. *The configuration file allows comments. They start with* `#` *and last til the end of the line. Empty lines are ignored.*

Note 3. *In general, the configuration file should be encoded in ASCII. The exception is encoding of values of options. Any file names should be encoded in the filesystem's encoding (they are not converted). The text-releated options (acronymsAfter, acronymsBefore) should have the same encoding as set in inputEncoding.*

1.1 Options types and default values

Allowed values for Boolean options are `yes`/`true`/`on`/`1` and `no`/`false`/`off`/`0`.

The default values, unless specified, are:

- for Boolean options - `no`
- for numeric options - `0`
- for string options - empty string

1.2 Files locations

All options in this group can contain a path, either relative (to the location of configuration file) or absolute (in Unixes starting from `/`, in Windows starting from `<letter>:\` or `\`).

1.2.1 tagset

The `tagset` option specified a file containing a tagset definition for processing. All the input data and the grammar must be consistent with this definition. This option *must* be present for any kind of execution of Spejd. The default is empty (so it will fail on every nonempty input data). For description of a syntax of this file see section 1.13.

example
```
tagset = sample-morfeusz.cfg
```

1.2.2 rules

The `rules` option specifies a file containing a grammar to be used by Spejd. It is needed only if the `spejd` tool is present in processing chain (the processing chain is explained in section 1.3.1). For description of a syntax of this file see chapter 3.

example
```
rules = rules.sr
```

1.3 Processing

1.3.1 processingChain

In the Spejd multiple tools can be executed between reader and writer modules. They form a processing chain, that is an output of a previous tool becomes an input of a next tool. The `processingChain` option lists names of subsequent tool that form the chain. Since the reader and writer are always used, they are not contained in the list. The names are separated by space. For standard Spejd distribution allowed values are:

- `spejd`
- `pantera` (not available in binary distribution of Spejd)
- `dictionary:<dictionary_name>`

The special notation for dictionary tool allows to apply multiple dictionaries in different places of processing chain. Each `<dictionary\_name>` suffix has to be unique in chain.

The default value of `processingChain` option is `spejd`. For details on the tools configuration see sections: 1.9 (spejd), 1.8 (pantera), 1.7 (dictionary).

```
example
processingChain = dictionary:example_dict spejd
processingChain = pantera spejd

# with an empty chain can Spejd act as a format converter
processingChain =
```

1.3.2 maxThreads

Most of the tools in Spejd can work in multiple threads on multiprocessor machines. The `maxThreads` option specifies how many threads (at most) can be used. The default is `0`, which causes the Spejd to detect the number of available processors/cores and use that many threads.

```
example
maxThreads = 4
```

Note 4. *Spejd doesn't split a single input file between threads, so there have to be multiple input files to get actual multithreading.*

1.4 Input data

1.4.1 inputType

The standard Spejd can read three input file formats:

- plain text
- XCES (as used in IPI PAN Corpus)
- TEI P5 (as used in NKJP - National Corpus of Polish)

The option `inputType` allows to specify the input type. Allowed values are `txt`, `xcesAna`, `tei` and `auto`, respectively for plain text, XCES, TEI P5 and an auto select mode. In the auto select mode the file type is detected basing on the file name/extension:

- plain text for *.txt/*.txt.gz

- XCES for morph.xml/morph.xml.gz
- TEI without using moprhosyntax information for ann_segmentation.xml / ann_segmentation.xml.gz
- TEI using morphosyntax for ann_morphosyntax.xml / ann_morphosyntax.xml.gz

The plain text reader and TEI reader without morphosyntax use Morfeusz morphological analyzer (consult section 1.10). When inputType is `tei` the expected input variant (segmentation/morphosyntax) is still determined by the file name. If the name contains "segmentation" string, the in ann_segmentation variant is used, for all other names the morphosyntax variant is used.

The default value for `inputType` is `auto`.

example
```
inputType = xcesAna
```

1.4.2 inputEncoding

The `inputEncoding` option defines the encoding of input files. It has to be set - Spejd doesn't use the XML coding tags.

The default value is `UTF-8`. Values acceptable are encoding names used by iconv, they can be listed by `iconv -l` command under Unixes. Using non ASCII-compatible encodings such as UTF-16 may be problematic, since this option affects some parts of the configuration file (the values of acronymsAfter and acronymsBefore options) as well as the Spejd grammar file.

example
```
inputEncoding = ISO8859-2
```

Note 5. *The complete list of strings/files decoded using this setting:*

- *configuration file: values of acronymsAfter and acronymsBefore options*
- *ogonkifier definition file*
- *dictionary files*
- *morfeusz disambiguation rules file*
- *quotes and non-keyword strings in grammar file*

1.4.3 inputFiles

The Spejd command line can contain paths to single input files or to directories. In the first case the specified files are simply processed. However if a directory is specified, Spejd searches it recursively for input files. The option `inputFiles` describes names of files to be processed. It has to be a regular expression.

The default value of `inputFiles` is an empty string. Hence, by default Spejd will process only files given explicitly in command line.

example

```
inputFiles = morph\.xml(\.gz)?|.*\.txt(\.gz)?|ann_morphosyntax\.xml(\.gz)?
```

1.4.4 ignoreDisamb

The Boolean `ignoreDisamb` option controls if Spejd has to use or ignore any disambiguation annotation found in input.

example

```
ignoreDisamb = no
```

1.5 Output

1.5.1 outputType

The standard Spejd can write two output file formats:

- XCES (as used in IPI PAN Corpus)
- TEI P5 (as used in NKJP - National Corpus of Polish)

The `outputType` option specifies which writer is to be used. It can be `xcesAna` or `tei` (for XCES and TEI P5 respectively) or `null` (a writer that actually does not write anything, it exists here for testing purposes).

The default is `xcesAna`.

example

```
outputType = xcesAna
```

1.5.2 discardDeleted

Spejd can either write deleted ('incorrect') interpretations to the output file or omit them (and write only those, which are decided to be 'correct'). The `discardDeleted` Boolean option controls that aspect of writer.

The default is not to discard.

```
example
discardDeleted = no
```

1.5.3 backupExistingFiles

When Spejd finds that the output file already exists, it can overwrite it or make a backup of the old file. This Boolean option controls if to rename old files to a ¡name¿.bak or simply overwrite them.

The default is to backup (`yes`).

```
example
backupExistingFiles = no
```

Note 6. *Spejd will not overwrite any file until the processing ends successfully. Instead a temporary file is created for the time of processing. Hence, Spejd can safely "overwrite" the files it is reading from.*

1.5.4 compressOutput

The `compressOutput` Boolean option controls whether the output has to be compressed using 'gzip' compression.

The default is `no`.

```
example
compressOutput = yes
```

1.5.5 compactTeiOutput

By the default, TEI writer produces files in format similar to National Corpus of Polish format, that is it writes empty sentences and paragraphs and places every xml tag in separate line. This Boolean option allows to produce more simple and more human readable output without empty sentences.

The default is `no`.

```
example
compactTeiOutput = yes
```

When turned off, an example fragment of morphosyntax file may look like:

```
example
<f name="orth">
 <string>uzależnione</string>
```

```
</f>
<f name="msd">
 <symbol value="pl:nom:m2" xml:id="morph_1.1.8.1.1-msd"/>
</f>
```

After turning it on, the above fragments look like this:

```
example
<f name="orth"><string>uzależnione</string></f>
<f name="msd"><symbol value="pl:nom:m2" xml:id="morph_1.1.8.1.1-msd"/></f>
```

1.5.6 teiSingleSyntokInterp

Spejd allows segments and syntactic words to be partially disambiguated. That means they can have more than one interpretation marked as "correct". In NKJP (National Corpus of Polish) format, it is illegal to have multiple "correct" interpretations for one token. This option allows to get full compatibility of TEI writer with National Corpus Of Polish, assuming, that grammar does not produce words with multiple "correct" interpretations. It causes writer not to output nonstandard XML-tags `<f name='interps'>` and `<fs type='lex'>` in `*_words.xml` files.

The default is `no`.

```
example
teiSingleSyntokInterp = yes
```

Note 7. *It is a user task to make sure, that there will be no tokens with multiple "correct" interpretations. If this option is set and a non-disambiguated token is found Spejd will terminate with an error message.*

1.5.7 teiFsGroupHeads

One more NKJP compatibility option for TEI writer. Causes the syntactic and semantic heads information of groups to be written as `<f name='...'` inside the group's feature structure (`<fs>`). When disabled, heads are marked as `type` attribute of group's elements.

The default is `no`.

```
example
teiFsGroupHeads = yes
```

When turned off, an example fragment of groups file may look like:

example
```
<seg xml:id="groups_1.1-s_3">
 <fs type="group">
  <f name="orth"><string>dwie decyzje</string></f>
  <f name="type"><symbol value="NumGz"/></f>
 </fs>
 <ptr type="synh" target="ann_words.xml#words_1.1-s_8"/>
 <ptr type="semh" target="ann_words.xml#words_1.1-s_9"/>
</seg>
```

After turning it on, the above fragment looks like this:

example
```
<seg xml:id="groups_1.1-s_3">
 <fs type="group">
  <f name="orth"><string>dwie decyzje</string></f>
  <f name="type"><symbol value="NumGz"/></f>
  <f name="synh" fVal="ann_words.xml#words_1.1-s_8"/>
  <f name="semh" fVal="ann_words.xml#words_1.1-s_9"/>
 </fs>
 <ptr target="ann_words.xml#words_1.1-s_8"/>
 <ptr target="ann_words.xml#words_1.1-s_9"/>
</seg>
```

1.5.8 teiBottomUpSyntacticStructures

Backward compatibility with Spejd 1.2. Sometimes it is easier to parse TEI when all entities are defined before they are referenced. This option allows to get such bottom-up order of entities (starting from leafs, ending with the root).

The default is `no`.

example
```
teiBottomUpSyntacticStructures = yes
```

1.5.9 outputSuffix

This option defines a suffix to be added to the output file name. It has to include the extension, but not the compression format suffix (if any).

The default is empty.

example
```
outputSuffix = Sh.xml
```

The example above will produce ”morphSh.xml” output file from ”morph.xml” input (if the outputType is xcesAna) or ”morph_morphosyntaxSh.xml”, ”morph_wordsSh.xml”, ”morph_namedSh.xml” and ”morph_groupsSh.xml” files for tei outputType (consult the next section for the ”morph” part of name). If the compression is turned on, the names will look like ”morphSh.xml.gz”.

For xml input files overwriting with output files one can use ”.xml”.

1.5.10 outputFilenameCore

To modify a core of file name the `outputFilenameCore` can be used. It causes Spejd to use a specified string instead of the input file name core while constructing the output file name(s).

If this option is empty or is not present in configuration file, Spejd will use the input file name core.

example

```
outputFilenameCore = ann
# this is a conventional name core for files in National Corpus of Polish
```

Note 8. *When using this option make sure there is no directory containing more than one file matching the* `inputFiles` *filter in the input. Otherwise the output files will be overwritten.*

1.6 Diagnostics

1.6.1 reportInterval

This option controls the time interval between each progress report in seconds. `0` value disables progress reports.

example

```
reportInterval = 5
```

1.6.2 debug

More detailed memory usage or efficiency debugging reports can be turned on using Boolean `debug` option. Setting it to `yes` also causes that a more detailed summary on Spejd finite-state machines usage is printed.

example

```
debug = no
```

1.6.3 ruleMarking

By default Spejd outputs rule titles in syntactic structures created by those rules, but disambiguation doesn't contain such information about rules. This option causes an additional attribute 'rule' containing rule title to appear in the output in each interpretation marked by Spejd as incorrect/deleted.

example
```
ruleMarking = yes
```

Note 9. *When* `ignoreDisamb` *is set to* `no`*, if an interpretation is marked as incorrect already in the disambiguated input, Spejd will not write the 'rule' attribute in the output for this interpretation. The* `ruleMarking` *affects only interpretations deleted by some Spejd rules.*

1.6.4 nonfatalTagErrors

By default, if some rule produce a tag not conforming the tagset, Spejd will terminate and write message about details of the error. If the `nonfatalTagErrors` is turned on, Spejd will not terminate on such errors. It will try to do its best to output only tags conforming the tagset, but they may be useless. This option exists only to preserve compatibility with older versions of Spejd, which accepted incorrect rules. Please do not use when developing new grammars.

Note 10. *Use this option at your own risk and don't report crashes when using it*

example
```
nonfatalTagErrors = no
```

1.6.5 muffleTagWarnings

If the `nonfatalTagErrors` is set, Spejd (probably) outputs large amount of warnings about incorrect tags. The `muffleTagWarnings` Boolean option disables printing them.

example
```
muffleTagWarnings = no
```

1.6.6 tagErrorsOnlyOnTheEnd

By default correctness checks of tags are performed on each tag modification. This option allows to disable all the checks during processing and leaves only one validation of tags before writing to the output file. Using this option is not recommended for developing new grammars.

example
```
tagErrorsOnlyOnTheEnd = no
```

1.7 Dictionaries

For each dictionary tool in the `processingChain` there must be one option specifying a list of files containing dictionary entries. The name of each of those options is the same as in the `processingChain`:
`dictionary:<dictionary\_name>`

example
```
# if dictionary:example_dict was the entry in 'processingChain', then
# two files 'sample_dict' and 'lexdictnum' can be assigned to it as follows

dictionary:example_dict = sample_dict lexdictnum
```

1.7.1 Syntax of the dictionaries

Each dictionary entry has to be in separate line. The entries should be in one of the following forms:

syntax
```
orthographic form,base (lexical) form:tag
```

syntax
```
,base (lexical) form:some_parts_of_tag;condition
```

In the first variant the orthographical form is used for matching words. Tag definition is expanded (it can contain wildcards).

In the second variant orthographic form is omitted. In that case a base form is used to match. The tags of existing interpretations which match the base form are corrected/modified according to the specified tag. This variant allows the tag to be not full/complete, but only specifying some of the attributes (some parts). This variant also allows to specify conditions on tag that must be meet to perform the modification. The condition has form of a partial tag, just like in the ”tag” section of modifying variant. A condition restricts modified interpretations to that ones which have all values of the specified attributes among the specified values. If an attribute is omitted in the specification it means that there are no restrictions on this attribute value and it can be anything (including absence of value). When a condition is empty (that means: there are no restrictions on any attribute), a semicolon preceding it can be omitted and the format is:

```
syntax
,base (lexical) form:some_parts_of_tag
```

Both variants of entries (with and without orthographic form) can be mixed. All entries with orthographic form are applied before applying any of the entries without orth in the scope of a single `dictionary:<name>` tool, no matter in which file in the file list they appear.

Here go some example dictionary entries

```
example
Korea Południowa,Korea Południowa:subst:sg:nom.voc:f
Korei Południowej,Korea Południowa:subst:sg:gen:f
Korei Południowej,Korea Południowa:subst:sg:dat:f
,ten:sen=17.8;nom.acc
,okrutny:sen=-13.1:rev;nom.acc
,gorzki:rev
```

The first three entries specify tags and bases for two-segment entry. Application of any of them will produce a syntactic word. The next two entries will match all interpretations of words with the bases 'ten' and 'okrutny' respectively which have case equal to 'nom' or 'acc'. They will modify their tags adding/setting numeric attribute 'sen' with appropriate value and (for the second entry) setting some other attribute to 'rev' (e.g. the 'rev' may be a value of reversibility attribute). The last entry will match all interpretations with the lexical form 'gorzki' and set their reversibility to 'rev'.

Note 11. *Wildcards in dictionaries are allowed, however in the modifying entries there is only 'one.two.three' format allowed (the '_' is illegal).*

Note 12. *The dictionary files should have the same encoding as the input files (which is set in the inputEncoding configuration option).*

1.8 Pantera configuration

1.8.1 panteraDoOwnMorphAnalysis

The Pantera can use its own built-in tweaked version of Morfeusz. If this option is set, all interpretations set by the reader or any tools preceding pantera in the `processingChain` are dropped before performing the Pantera analysis.

```
example
panteraDoOwnMorphAnalysis = yes
```

1.8.2 panteraTagsetName

This option specifies the tagset to be used by Pantera. Leave empty to use Pantera's default. This value directly sets the Pantera's 'tagset' option, for details refer to the Pantera documentation.

example
```
panteraTagsetName = ipipan
```

Note 13. *It is important to make sure the tagset definition specified by 'tagset' option (section 1.2.1) is a superset of the tagset used by pantera.*

1.8.3 panteraEnginePath

This option specifies the engine to be used by Pantera. Leave empty to use Pantera's default. This value directly sets the Pantera's 'engine' option, for details refer to the Pantera documentation.

example
```
panteraEnginePath = my_engine.btengine
```

Note 14. *Relative paths are interpreted as relative to the location of the 'config.ini' file.*

1.9 Spejd semantics and internals configuration

1.9.1 matchStrategy

This option sets the strategy for matching syntactic entities. It can be either `*` for greedy, `+` for possessive or `?` for reluctant. The default is `*`.

example
```
matchStrategy = *
```

1.9.2 nullAgreement

This Boolean option controls whether 'agree' or 'unify' operations should return true if none of the agreed entities has the agreed attribute. In other words if agree(case,1,2) should return true, if both entity 1 and 2 have no case? Default is no.

example
```
nullAgreement = no
```

1.9.3 composeLimit

This integer option defines a number of single-rule automata to be composed together. Usually there is no need to change the default value (it has been chosen to obtain the maximal efficiency).

Note 15. *Rule of thumb: if Spejd consumes much too much memory, it's better to decrease this number than to set the* `memoryLimit` *option to a low value - it gives smaller impact on performance while significantly decreasing the amount of memory used by Spejd.*

example

```
composeLimit = 150
```

1.9.4 memoryLimit

This integer option defines a memory limit in megabytes. When memory usage exceeds this limit a rarely-used states removal procedure is launched. This option should be rather used as an emergency brake. For a significant reduction of the memory usage consider changing the `composeLimit` option.

The default is 1000.

example

```
memoryLimit = 2000
```

Note 16. *There is no guarantee that Spejd will not consume more memory than specified by memoryLimit. The limit is treated as a hint when to start to conserve memory. Actual usage may be higher by few percents (it depends on memory allocator library buffers size).*

Note 17. *For 32-bit systems/binaries: values higher than approx. 1900 are useless and may cause Spejd terminate when trying to allocate more memory that is possible in 32-bit systems (2048 megabytes).*

1.9.5 leavePercent

This integer option specifies approximate percent of FSM states to leave after the states removal. For best effects use values between 50 and 90 depending on diversity of the processed data and used `memoryLimit` (lower values work better with high diversity of natural language in subsequent input files, high value may cause the states removal procedure to start frequently wasting lots of time).

Use values between 1 and 100.

example

```
leavePercent = 80
```

1.9.6 minComplexPercent

The definitive limit of normal state removal procedure usage. State removal deletes only complex states, so if there are lots of plain states it can't prevent from exceeding `memoryLimit`. If the percent of complex states is less than `minComplexPercent` of all states, all the DFAs are dropped and they are built from the beginning just like if the spejd would be restarted. However it does not recompile rules, so it's faster.

```
example
minComplexPercent = 10
```

1.9.7 maxNumberOfValues

This integer option specifies a maximal number of unicode characters which can appear in rules compiled to internal regex. It must be higher than the highest number of values of a single attribute (including numeric attributes) and must be higher than a number of unique characters appearing in all rules. Setting it to high values can increase the memory usage.

Default is 4000.

```
example
maxNumberOfValues = 4000
```

1.10 Morfeusz (morphological analyzer) configuration

1.10.1 disableMorfeusz

Use this option to disable Morfeusz completely. It can be useful when some tool in the processingChain (e.g. pantera) replaces interpretations produced by the reader.

```
example
disableMorfeusz = no
```

1.10.2 morfeuszSegmentationDisambiguationRules

Morfeusz produces ambiguous segmentation, which is not allowed in Spejd. It can be resolved by a simple rule-based disambiguator. This option specifies a file to load rules from. The default (empty value) causes to use few built-in rules for Polish.

example
```
morfeuszSegmentationDisambiguationRules = segm_disamb.conf
```

The format of file is as follows:

- each rule is in a separate line
- each rule consists of 2 words separated by space(s)
- the first word is a regexp pattern matching the orthographic form of the word
- the second word is a keyword (one of `separate` and `together`)
- comments start from `#` character and terminate with newline
- empty lines are allowed

The default rules are:

example
```
baliście separate
czekał.m separate
kulturalno-oświatowy.* separate
miałem separate
piekłem separate
podziałom together
winnym together
wyłom together
.*łem together
.*ś together
```

Note 18. *Encoding of this file must be as set in the inputEncoding option.*

1.11 Plain text reader

1.11.1 stringRangeMockID

The plain text reader marks position in the input text of each word in a string-range notation. It requires a 3-rd attribute which identifies an element in which the position is calculated. This option sets a mock identifier for this purpose.

example
```
stringRangeMockID = p-1
```

1.11.2 acronymsAfter

This option specifies a list of acronyms. If a dot is found after one of them, the sentencer doesn't consider it as a sentence break. The list is separated by `|`.

```
example
acronymsAfter  = prof|dr|mgr|doc|ul|np|godz|gen|płk|mjr|por|tzw|tzn|proc|nt|art|ust|ww|www|
```

1.11.3 acronymsBefore

This option specifies a list of another kind of acronyms (usually top level domain names). If a dot is found before one of them, the sentencer doesn't consider it as a sentence break. The list is separated by `|`.

```
example
acronymsBefore = ac|ad|ae|aero|af|ag|ai|al|am|an|ao|aq|ar|arpa|as|asia|at|au|aw|ax|az
```

1.11.4 ogonkifyFile

This option sets a name of a file containing ogonkify (diacritic completion) substitutions.

```
example
ogonkifyFile     = ogonkifier.ini
```

The format of the file is:

```
syntax
<letter without diacritics>=<list of possible letters with diacritics separated by '|'>
```

```
example
s=ś
z=ź|ż
```

Note 19. *This option is required when using plain text reader unless the ogonkifier is disabled by* `ogonkifyStrategy` *option.*

1.11.5 ogonkifyStrategy

This option controls when the ogonkifier is used. It can have one of the values:

- `A` - use always
- `N` - never (disables ogonkification)
- `M` - only when the morphological analyzer fails to analyse a word

```
example
ogonkifyStrategy = M
```

1.11.6 ogonkifyMinLength

This option specifies minimal length of a word to be processed by ogonkifier.

```
example
ogonkifyMinLength = 3
```

1.11.7 ogonkifyMinLength

This option specifies maximal length of a word to be processed by ogonkifier.

```
example
ogonkifyMaxLength = 13
```

Note 20. *The ogonkifier produces every possible combination of diacritic completed and not completed characters that appear in word. The number of them can be exponential on the word length.*

1.12 Additional configuration

Spejd is designed in the way that additional extensions/tools can be written easily. They can use some additional configuration options which should appear in the configuration file. If you are using such a nonstandard extensions consult their documentation for details.

1.13 Tagset syntax

As in the configuration file, in the tagset file comments are allowed. They start with # and last til the end of the line. Empty lines are ignored. The encoding of this file must be ASCII (this doesn't apply to comments as long as they are well formed).

The tagset definition file should consist of two sections: ATTR and POS, in this order.

Note 21. *Names and values of attributes and parts of speech must be single words containing letters and underscores. They are case-sensitive, they must not be single capital letter and must not be equal to any of the rules syntax keywords.*

Note 22. *Number of values of a single attribute (including numeric attributes) must not exceed* `maxNumberOfValues` *defined in configuration.*

1.13.1 Attributes

First section of the tagset definition file is `[ATTR]`. It lists attributes and their values. For enumerable attributes the syntax is:

syntax
```
name_of_attribute = first_value second_value third_value ...
```

The numeric attributes are defined using the following syntax

syntax
```
attrname  =  <lower_bound, upper_bound> number_of_partitions
```

`lower_bound` and `upper_bound` are floating point numbers and optional `number_of_partitions` is an integer. All the numeric values of the attribute will be stored rounded to a multiple of $precision = (upper\_bound - lower\_bound)/(number\_of\_partitions - 1)$. That means they will have form of $lower\_bound + n * precision$ with $n \in [0, number\_of\_partitions)$

The default value of `number_of_partitions` is $round(upper\_bound - lower\_bound) + 1$, so if both bounds are integers the $precision$ equals to 1.

The `number_of_partitions` must not be lower than 2 and should not exceed `maxNumberOfValues` defined in configuration. Setting it to high values can cause performance reduction and higher memory usage.

example
```
# the section header
[ATTR]
```

```
# here goes the list of attributes with values
# some enumerable attributes
number                  = sg pl
case                    = nom gen dat acc inst loc voc
gender                  = m1 m2 m3 f n n1 n2 n3 p1 p2 p3
person                  = pri sec ter
degree                  = pos com sup
aspect                  = imperf perf
negation                = aff neg

# and some numeric attributes
sen                     = <-20,20> 401
percent_attribute       = <0,100>
# the last one has steps of size 1, so it has 101 values
```

1.13.2 Parts of speech

The second section consists of part of speech definitions with lists of possible attributes. Optional attributes are marked with square brackets.

example

```
# the section header
[POS]

# here go the part-of-speech definitions
conj    =
interp  =
adv     = [degree] [sen]
imps    = aspect [negation] [sen]
subst   = number case gender [sen]
ger     = number case gender aspect negation
```

Note 23. *Number of POS-es should not exceed 'maxNumberOfValues' defined in configuration.*

Chapter 2

Input and output formats

IO formats used by Spejd are not well defined elsewhere, so it is worth to describe them in details. Example files of all variants of formats discussed in this chapter are attached to the Spejd package, in the `examples/format_examples.tar.gz` archive.

2.1 Input only

2.1.1 Plain text

The simplest input format is plain text. It has to be have correct encoding (according to settings in configuration file). The whole text file is considered to be a single paragraph, multiple blank characters (new lines, spaces) are treated as a single white space. The text is processed by a simple sentencer and tokenizer, so please remember to configure `acronymsBefore` and `acronymsAfter` configuration options to get what you expect. By default, tokenized text is then processed by the built-in morphological analyzer (Morfeusz). See section 1.10 for analyzer configuration, especially if you want to disable it.

2.2 Input-output formats

2.2.1 xcesAna

The "xcesAna" supported by Spejd is similar to the format used by old versions of Spejd (0.8x). It is similar to a modification of the original "Corpus Encoding Standard Encoding conventions for annotated data" for the IPI PAN Corpus [1] with some minor changes. Additionally Spejd supports a syntactic entities notation (`group`, `syntok` tokens). The "xcesAna" format

[1]dtd for the format is available at http://korpus.pl in a download section, along with the "Frequency dictionary of contemporary Polish"

is (so far) the only format which is lossless for the information that Spejd uses and produces. That means, the output of Spejd in this format can be read again without loss of information produced by a previous processing.

Below there is a structure of the ”xcesAna” format explained in details with tiny examples.

header

```
example
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE cesAna SYSTEM "xcesAnaIPI.dtd">
```

The above header has to be in the exactly this form. The `version` and `encoding` attributes of `<?xml .. ?>` token are ignored, the encoding is set through configuration file.

base file structure

```
example
<cesAna xmlns:xlink="http://www.w3.org/1999/xlink" type="pre_morph" version="IPI-1.2">
<chunkList xml:base="text.xml">
<!-- chunks follows here -->
</chunkList>
</cesAna>
```

The whole file consists of a `cesAna` token containing exactly one `chunkList`. There can be any kind of attributes in that tokens, all are ignored while reading, but they are saved and written in the output in corresponding tokens (if only the output format is xcesAna too).

chunks, text structure

```
example
<chunk type="caption" xlink:href="#caption1">
<chunk type="s">
<!-- list of entities follows>
</chunk>
</chunk>
<chunk type="p">
<chunk type="s">
<!-- list of entities follows>
</chunk>
<chunk type="s">
<!-- list of entities follows>
</chunk>
</chunk>
```

Chunks build the file structure. Spejd operates at the level of sentences, so the list of chunks must at least reflect the sentences structure in the text. However chunks can be nested, so there is a possibility to render more complicated structure and meta information as paragraphs, titles, headers, footnotes. Spejd distinguishes sentence-chunks from other chunks by using `type` xml attribute, which must be equal to `"s"` in the case of sentence. If there is no chunk of type `"s"`, the most nested chunk is treated as sentence. The sentence chunk contains list of entities and must not contain any nested chunks; if the chunk contains a nested chunk, it must neither contain any entities nor have a type `s`. A correctly built chunks structure is copied to the output with all their token attributes if the output is xcesAna. In the "no type-s chunk" case, an additional chunks level is added to the output marking the sentences with the "`type="s"`".

entities

An example of annotated text without syntactic structures:

```
example
<tok>
<orth>Niepozorne</orth>
<lex><base>niepozorny</base><ctag>adj:pl:nom:m2:pos</ctag></lex>
<lex disamb="1"><base>niepozorny</base><ctag>adj:pl:nom:m3:pos</ctag></lex>
<lex><base>niepozorny</base><ctag>adj:pl:acc:n:pos</ctag></lex>
</tok>
<tok>
<orth>skarby</orth>
<lex disamb="1"><base>skarb</base><ctag>subst:pl:nom:m3</ctag></lex>
<lex><base>skarb</base><ctag>subst:pl:acc:m3</ctag></lex>
</tok>
<ns/>
<tok>
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
<ns/>
<tok>
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
<ns/>
<tok>
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
```

The same with syntactic structures:

example

```
<group id="a3" rule="(1) NG between verbs/groups/aby/etc." type="NG" synh="a2" semh="a2">
<tok id="a1">
<orth>Niepozorne</orth>
<lex disamb_sh="0"><base>niepozorny</base><ctag>adj:pl:nom:m2:pos</ctag></lex>
<lex disamb="1"><base>niepozorny</base><ctag>adj:pl:nom:m3:pos</ctag></lex>
<lex disamb_sh="0"><base>niepozorny</base><ctag>adj:pl:nom:f:pos</ctag></lex>
</tok>
<tok id="a2">
<orth>skarby</orth>
<lex disamb="1"><base>skarb</base><ctag>subst:pl:nom:m3</ctag></lex>
<lex disamb_sh="0"><base>skarb</base><ctag>subst:pl:acc:m3</ctag></lex>
</tok>
</group>
<ns/>
<syntok id="a11" rule="...">
<orth>...</orth>
<lex><base>...</base><ctag>interp</ctag></lex>
<tok id="ab">
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
<ns/>
<tok id="ad">
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
<ns/>
<tok id="af">
<orth>.</orth>
<lex disamb="1"><base>.</base><ctag>interp</ctag></lex>
</tok>
</syntok>
```

An entity can be one of four types:

- segment - a single string of characters with morphological information attached (a single word). It is represented by `<tok>` xml token. There are no required attributes this token. It must have an `<orth>` subtoken and may have one or more `<lex>` subtokens (interpretations).
- a no-space - token marking that there is no whitespace between subsequent segments. Represented by `<ns/>` xml token. The no-space has no required attributes and must not contain subelements. All existing attributes are ignored.
- a syntactic word - a sequence of segments, nested syntactic words and no-spaces with its morphological information. Represented by

<syntok> xml token. There are no required attributes of this token. In the output of Spejd, there can be a rule attribute with a name of a rule which has built this synthetic word. Like segment, a syntactic word must have an <orth> subtoken and may have one or more <lex> subtokens. Additionally it must have one or more <tok>, <syntok> or <ns/> subtokens.

- a syntactic group - a sequence of segments, no-spaces, syntactic words and groups with specific type information and a chosen head entities. Represented by <group> token. The <group> token must have a type, synh and semh attributes and optionally may have a base attribute. The synh and semh attributes contain identifiers of, respectively, syntactic and semantic head. It is required, that the entities representing those heads have id xml attributes with the same identifiers. In the output of Spejd, there can be a rule attribute with a name of a rule which has built this synthetic group. A group must have one or more subentities (of any of the four types).

In the output of the Spejd all segments, syntactic words and groups have id attributes with their own, unique values (in the scope of the file).

In the list above, the <orth> xml token represents orthographic form of the entity. It may not have any attributes and should be a text-only token (with no subtokens, only plain text).

The <lex> token may have a disamb attribute with a value 1. That means the interpretation represented by this token is a chosen one (having "golden tag"). In the output, the negative choices made by Spejd are marked with disamb_sh attribute with a value 0 (those interpretations are "deleted" by Spejd grammar). Additionally, depending on configuration, "deleted" interpretation may have a rule attribute which contains a name of a rule which has deleted the interpretation. Any other attributes are ignored and copied to the output (as long as the output is xcesAna). The <lex> should have exactly two text-only subtokens: <base> and <ctag>, the first containing a base (lexical) form of word(s) and the second containing a positional tag, with POS and other attribute values separated by colon.

2.2.2 TEI

The TEI format supported by Spejd is a TEI P5 encoding similar to the encoding used in National Corpus of Polish (NKJP) [2]. The format uses multiple files to encode various layers of annotation. Spejd can read segmentation (optionally referring to a text layer) and morphosyntactic annotation layers and is able to write segmentation, morphosyntactic, syntactic words and syntactic groups layers. The format of files written by Spejd may vary

[2]Schemes for this format can be found at http://nlp.ipipan.waw.pl/TEI4NKJP/

slightly depending on few switches set in configuration listed at the end of this section. In particular it can be fully compatible with NKJP (as of May 2012).

The segmentation and morphosyntactic layers are parsed by dedicated parsers, so their formats are hardly flexible. The text layer is read using xml library, thus reading is not so efficient as reading other layers but it is highly flexible. Spejd doesn't support reading multiple layers at the same type (except for text layer referenced from segmentation). Hence, the TEI format is not lossless for Spejd, it cannot read syntactic structures from its output. For deciding which layer (segmentation or morphosyntax) is read when both are available see section 1.4.1.

text layer

This level is not read or written by Spejd directly. It is used when the segmentation layer doesn't contain orthographical forms and only refers to particular parts of text. There are no restrictions on the format of this layer except it has to be either a plain text file with the name ending with `.txt` or a valid xml.

base structure of the remaining layers files

example

```
<?xml version="1.0" encoding="UTF-8"?>
<teiCorpus xmlns:xi="http://www.w3.org/2001/XInclude"
    xmlns="http://www.tei-c.org/ns/1.0" xmlns:nkjp="http://www.nkjp.pl/ns/1.0">
 <xi:include href="NKJP_1M_header.xml"/>
 <TEI>
  <xi:include href="header.xml"/>
  <text>
   <body>
    <p> <!-- paragraph -->
     <s> <!-- sentence -->
       <!-- contents (segments, syntactic structures) go here -->
     </s>
     <s> <!-- another sentence -->
       <!-- contents (segments, syntactic structures) go here -->
     </s>
    </p>
   </body>
  </text>
 </TEI>
</teiCorpus>
```

The segmentation, morphosyntactic, syntactic words and groups layers use the same structure in their files.

The only optional tokens in the example above are the `<xi:include` ones. They are ignored by Spejd, but saved and written to the output (as long the output is TEI). Replacing them by contents of the referred files is not supported. That are the only places where `<xi:include` tags may be present. The encoding set in the `<?xml` token is ignored, user should set the correct encoding in the Spejd's configuration file. All attributes of `<teiCirpus>`, `<TEI>`, `<text>` and `<body>` are ignored but saved and written to the output. Inside the `<body>` token should be a list of `<p>` tokens (paragraphs). It may be empty. Each paragraph consist of a list of sentences (`<s>` tokens). Possible contents of `<s>` differ between layers and are described below.

segmentation layer

example
```
<!-- 1st version, with orth forms in a separate file -->
  <choice>
  <nkjp:paren>
  <!-- Gdzie -->
  <seg corresp="text.xml#string-range(p1,0,5)" nkjp:rejected="true"
       xml:id="segm_seg1"/>
  <!-- ś -->
  <seg corresp="text.xml#string-range(p1,5,1)" nkjp:nps="true"
       nkjp:rejected="true" xml:id="segm_seg2"/>
  </nkjp:paren>
  <nkjp:paren>
  <!-- Gdzieś -->
  <seg corresp="text.xml#string-range(p1,0,6)" xml:id="segm_seg3"/>
  </nkjp:paren>
  </choice>
  <!-- tam -->
  <seg corresp="text.xml#string-range(p1,7,3)" xml:id="segm_seg4"/>
  <!-- ? -->
  <seg corresp="text.xml#string-range(p1,11,1)" nkjp:nps="true"
       xml:id="segm_seg5"/>
```

example
```
<!-- 2nd version, with explicitely given orth forms -->
  <seg xml:id="segm_seg6">Tak</seg>
  <seg xml:id="segm_seg7">.</seg>
```

Segmentation layer consists of `<seg>` (segment) tokens. They are expected to form a list (subsequent tokens in a sentence) with exceptions for fragments with ambiguous segmentation. Those fragments are closed in `<choice>...</choice>` token and may be grouped using `<nkjp:paren>` tokens. The `<nkjp:paren>` parentheses should not be nested.

Every segment has to point to the text file to its orthographic form using `corresp` attribute and `string-range` function (it has 3 arguments, id of xml text-token, starting position counting from 0 and length of the substring; the id is ignored when referring to a plain text file). Alternatively, the orthographic form may be included directly as contents of the `<seg>` token as in the "2nd version" in the example above. Output segmentation files produced by Spejd (created only when the input is plain text) use the first version, referring to the input text file.

Segments may optionally have `nkjp:nps` attribute set to `true` (there is no white-space before segment), and `xml:id` identifier unique in the scope of file. Ambiguities `<choice>` should be resolved, the notation for marking rejected variants is `nkjp:rejected` attribute present in every segment in that variant.

morphosyntax layer

example

```
<seg corresp="ann_segmentation.xml#segm_1.8-seg" xml:id="morph_1.1.8-seg">
 <fs type="morph">
  <f name="orth"><string>uzależnione</string></f>
  <f name="interps">
   <fs type="lex" xml:id="morph_1.1.8.1-lex">
    <f name="base"><string>uzależniony</string></f>
    <f name="ctag"><symbol value="depr"/></f>
    <f name="msd"><symbol value="pl:nom:m2" xml:id="morph_1.1.8.1.1-msd"/></f>
   </fs>
   <fs type="lex" xml:id="morph_1.1.8.2-lex">
    <f name="base"><string>uzależnić</string></f>
    <f name="ctag"><symbol value="ppas"/></f>
    <f name="msd">
     <vAlt>
      <symbol value="sg:nom:n:perf:aff" xml:id="morph_1.1.8.2.1-msd"/>
      <symbol value="pl:acc:n:perf:aff" xml:id="morph_1.1.8.2.2-msd"/>
      <symbol value="pl:nom:f:perf:aff" xml:id="morph_1.1.8.2.3-msd"/>
     </vAlt>
    </f>
   </fs>
  </f>
  <f name="disamb">
   <fs type="tool_report">
    <f fVal="#morph_1.1.8.1.1-msd" name="choice"/>
   </fs>
   <fs type="tool_report">
    <f fVal="#morph_1.1.8.2.1-msd" name="choice"/>
    <f name="interpretation"><string>uzależnić:ppas:pl:nom:n:perf:aff</string></f>
   </fs>
   <fs type="tool_report">
    <f fVal="#morph_1.1.8.2.3-msd" name="choice"/>
   </fs>
  </f>
 </fs>
</seg>
```

The morphosyntax layer consists of a list of segments. The example above shows typical contents of a single `<seg>` token (segment) in the morphosyntax file. The `<fs type="morph"` token with `<f name="orth">` and `<f name="interps">` subtokens are required. Lex entries list (`<fs type="lex">` tags inside the `<f name="interps">`) may be empty. Each lex entry has to have nonempty base form, a ctag (part of speech) and at least one morphosyntactic description msd. Multiple msds are possible within one lex entry and should be closed in `<vAlt>` token. Each msd with corresponding base and ctag form an interpretation (in Spejd nomenclature). The format of interpretations is exactly like in example. The required xml attributes are: `value` attributes of `<symbol>` tokens, `name` of `<f>` tokens and `xml:id` of `<symbol>` encoding values of msds. `xml:id` of other tags are optional. All other attributes are ignored by Spejd, but preserved in the output (if the output format is tei).

The last feature `<f name="disamb">` (disambiguation) is optional. It must contain at least one `<fs type="tool_report">` token and a `<f name="choice">` token pointing to a chosen interpretation/msd. The `<f name="interpretation">` is optional and not used by Spejd, but is supported and written in the output according to results of Spejd processing.

The binary format of this layer in the output may be changed using compactTeiOutput option (see 1.5.5). Turning it off causes subtokens of the inner `<f>`s to be written in a separate line each.

syntactic words layer

example

```
<!-- a single segment, directly corresponding with morphosyntax layer -->
<seg xml:id="words_1.1.8-seg">
 <fs type="words">
  <f name="orth"><string>uzależnione</string></f>
  <f name="interps">
   <fs type="lex">
    <f name="base"><string>uzależniony</string></f>
    <f name="ctag"><symbol value="depr"/></f>
    <f name="msd"><symbol value="pl:nom:m2"/></f>
   </fs>
   <fs type="lex">
    <f name="base"><string>uzależnić</string></f>
    <f name="ctag"><symbol value="ppas"/></f>
    <f name="msd">
```

```
      <vAlt>
       <symbol value="pl:nom:f:perf:aff"/>
       <symbol value="pl:nom:n:perf:aff"/>
      </vAlt>
     </f>
    </fs>
   </f>
  </fs>
  <ptr target="ann_morphosyntax.xml#morph_1.1.8-seg"/>
 </seg>
```

example

```
<!-- a true syntactic word -->
<seg xml:id="words_2.4-s_86"><!--  rule="negacja dla czasowników, imiesłowów" -->
 <fs type="words">
  <f name="orth"><string>nie ma</string></f>
  <f name="interps">
   <fs type="lex">
    <f name="base"><string>mieć</string></f>
    <f name="ctag"><symbol value="Verbfin"/></f>
    <f name="msd"><symbol value="sg:ter:pres:ind:imperf:nrefl:neg"/></f>
   </fs>
  </f>
 </fs>
 <ptr target="ann_morphosyntax.xml#morph_2.4.4-seg"/>
 <ptr target="#words_2.4-s_87"/>
</seg>
```

The syntactic words layer contains segments and syntactic words. For every segment in morphosyntactic layer there is a separate `<seg>` token. Additionally, every syntactic word is represented with its own `<seg>`. Each `<seg>` contain a `<fs type="words">` defining the orthographical form and interpretations. The format of interpretations is the same as in the morphosyntax layer, however there is no need for `xml:id` attributes in msds since this layer contains only interpretations considered to be correct. After `<fs type="words">` there go a list of `<ptr>` tokens pointing to the contents of syntactic word: to the corresponding segments in morphosyntax layer or to the nested syntactic words.

It is possible to change the contents of `<fs type="words">` using teiSingleSyntokInterp option. It causes `<f name="interps">` and `<fs type="lex">` tokens to be omitted and the base, ctag and msd `<f>`s to be written directly inside `<fs type="words">`. This option exists for compatibility reasons, see 1.5.6 for details.

The default behavior is to write the nested syntactic words in top-down order, starting with the root and ending on leafs. This can be changed using teiBottomUpSyntacticStructures option, see 1.5.8 for details.

The binary format of this layer in the output may be changed using compactTeiOutput option (see 1.5.5). Turning it off causes subtokens of the inner `<f>`s to be written in a separate line each.

syntactic groups layer

example

```
<seg xml:id="groups_1.2-s_10"><!--  rule="PrepNG: Prep + NG" -->
 <fs type="group">
  <f name="orth"><string>dla Jana Kowalskiego</string></f>
  <f name="type"><symbol value="PrepNG"/></f>
 </fs>
 <ptr type="synh" target="ann_words.xml#words_1.2-s_33"/>
 <ptr type="semh" target="#groups_1.2-s_11"/>
</seg>
<seg xml:id="groups_1.2-s_11"><!--  rule="NG: name + surname" -->
 <fs type="group">
  <f name="orth"><string>Jana Kowalskiego</string></f>
  <f name="base"><string>Kowalski, Jan</string></f>
  <f name="type"><symbol value="NG"/></f>
 </fs>
 <ptr type="head" target="ann_words.xml#words_1.2-s_34"/>
 <ptr type="nonhead" target="ann_words.xml#words_1.2-s_35"/>
</seg>
```

The syntactic groups layer consists of (only) syntactic groups - it doesn't refer anyhow to segments or words not closed in a syntactic group. Each `<seg>`, standing for a single group, must contain a feature set `<fs type="group">` describing the group and a list of pointers to its contents. The description consists of an orthographical form and a type, which is an arbitrary identifier provided by a grammar developer. It can also contain an optional feature `base`. Pointers to contents of the group have a `type` attribute with one of values: `nonhead`, `head`, `semh` or `synh`, meaning respectively, that this entity isn't a head of any type, is both semantic and syntactic head, is a semantic head only or is a syntactic head of the group only.

The notation of marking of heads by attributes of pointers can be changed to be included in the `<fs type"group">` token. See 1.5.7 for details.

The default behavior is to write the nested syntactic groups in top-down order, starting with the root and ending on leafs. This can be changed using teiBottomUpSyntacticStructures option, see 1.5.8 for details.

The binary format of this layer in the output may be changed using compactTeiOutput option (see 1.5.5). Turning it off causes subtokens of the inner `<f>`s to be written in a separate line each.

Chapter 3

Spejd grammar

The Spejd grammar consists of a single file. It has three main parts: variable definitions, macro definitions and rules, exactly in this order. The variable and macro definitions are optional.

In the whole file comments begin with `#` character and end with a newline character. Any whitespaces in the file are ignored (unless they are in quotes). Only ASCII double quotes (`"`) make quoted strings. The double quote character can appear in the quoted string, but must be backslash-escaped (`\"`). A backslash must also be escaped (`\\`). Any other characters in quotes are interpreted as they are.

Note 24. *The grammar file should be in an ASCII-compatible encoding. All keywords and braces/operators must be the same as in ASCII. Quotes contents (requirements on orth and base attributes) should be in the encoding set by the inputEncoding configuration option.*

3.1 Terminology

In this chapter each occurrence of *segment*, *token*, *syntactic word*, *syntactic group* and *syntactic entity* should be understood as follows:

- *segment* - a smallest interpreted unit, a sequence of characters together with its morphosyntactic interpretations (lemma, grammatical class, grammatical categories). In many cases it is a single word.
- *syntactic word* - a non-empty sequence of segments and/or syntactic words, e.g. named entity. It also has morphosyntactic interpretations.
- *token* - a segment or a syntactic word.
- *syntactic group* - a non-empty sequence of tokens and/or syntactic groups. It is identified by a syntactic head and a semantic head, which both are tokens. It has a type identifier.

- *syntactic entity* - a token or a syntactic group.

3.2 Variable definitions

In the top of the grammar file variables may be declared and defined. Declarations are constructed with one of the keywords: `Variable` or `ReportedVariable`, followed by variable name. After the name there can be a default value definition: an equality sign and a constant expression, in most cases a floating point number (see numeric expressions description for details, section 3.5.1). If the value definition is absent, the default value is set to 0.0 . Declaration must end with semicolon. The only difference between ReportedVariable and Variable is that the last value (from the last assignment to the variable during a single sentence processing) of ReportedVariable is put into the output file, while Variable is accessible only within grammar. Depending on the file format it is written as an attribute in the sentence tag or in a comment.

example

```
# An example of a reported variable declaration, without
# a default value definition (= default is 0.0).
ReportedVariable total_sentiment;

# another example variable, this will not appear
# in the output file and has a default value definition
Variable other_variable = 3/2;
```

All variables have limited lifetime: their values are reset to default before processing each sentence.

The variables can be accessed from rules by their names preceded by `@` character.

example

```
Rule "example"
Match: [sen>0];
Eval: assign(@total_sentiment, @liczbas_counter + 1.sen);

# in this example the total_sentiment is increased by
# a value of sen attribute from # the matched word
# (matched are all words with sen > 0).
```

3.3 Macro definitions

The macro definitions are constructed with keyword `Define` followed by macro name, equality sign and the macro value. They must end with semicolon. They should consist of one or more syntactic entity specifications (see section 3.4.2 for details on specifications syntax). They will be later used with name preceded by `$` sign. They can be used in rules and also within subsequent macro definitions.

example

```
# a simple definition
Define czasownik = [pos~~"winien|praet|bedzie|fin|impt|imps|inf"];

# a more complex definition using a previously defined macro
Define czasownik_lub_imieslow =
    [pos~~"winien|praet|bedzie|fin|impt|imps|inf|pcon|pant|pred"]
    | $czasownik;
```

example

```
Rule "example"
Match: [orth~"[Nn]ie"] \$czasownik;
Eval: word(3, neg, "nie " base);

# an example of macro usage in a rule
```

3.4 Rules

Every rule must start with a keyword `Rule` followed by rule title in double quotes. It may be not unique or empty - it's only for simplification of grammar development. Then goes a match specification followed by a list of operations to perform on the text.

The match specification is split into sections. Each section starts from a keyword followed by colon, consists of a regular expression over entity specifications ended by semicolon. This regexp can contain sequences, alternatives (in parentheses, separated by `|`) and quantifiers (`*`, `+` and `?`) like typical regular expressions.

The available sections are: `Between`, `Left`, `Match` and `Right` (that names are also the keywords starting each section). The `Match` section is a main part of match specification - all the operations should be performed on the entities matched by this part. Furthermore, the syntactic structures are built from all entities matched by this section. The `Left` and `Right` sections specify left and right context of a match. The `Between` section specifies a sequence of entities which may occur between any pair of specification units in `Match` or contexts. It can be for example a sigh or a pause in a spoken language. Only `Match` section is required.

The list of operations to be performed on the matched text starts from the keyword `Eval` followed as usual by colon. Every operation has form:

syntax

```
keyword(some, arguments);
```

Operations can fail, in which case execution of the list is stopped (like for the predicates in Prolog, however any changes made by previous operations are not reverted).

In all the operations references to specification units can be either numbers of units (counting from 1, not considering the contents of the Between section) or defined in match specification capital letters.

All available operations are described in section 3.5.

example

```
# a complex example

Rule "Uncertain: NG with genitive postmodifier between verbs/groups/aby/etc."

Between: [orth~"\.\.\."];
Left:    (sb | $lub_grupa | [orth~","]);
Match:   [orth~"[Jj]ego|[Jj]ej|[Ii]ch"]?
         ([orth~"tzw"] ns [orth~"\."])?
        A[pos~"adj|pact|ppas"]* B[pos~~"subst|ger"]
        C[pos~"adj|pact|ppas" && case~"gen"]*
        D[pos~~"subst" && case~"gen"] ;
Right:   (ns? [pos~"interp"])* [pos~~"qub|adv|conj"]*
         ([synh=[]] | ns? [orth~","]? [orth~"że|żeby|aby|by"] |
          (ns? [pos~"interp"])* se | ns? [orth~","]);
Eval:    unify(case number gender,A,B);
         leave(case~~"gen",D);
         unify(case number gender,6,7);
         group(NG,5,5);
         assign(@total_sentiment = @total_sentiment + B.sen);

# references above are: A = 4, B = 5, C = 6, D = 7
# note the escaped dots in regexp in Between section
# (we want to match "...")
```

3.4.1 Specification units

Every specification unit (that is: single entity specification with optional quantifier or an alternative closed in parenthesis) can by preceded by a capital letter, which can be easy referred to by operations. The letters must be unique in a whole rule. For example, for the following match specification:

example

```
Left: sb?
Match: A[orth~"[1-9][0-9]*"] B(ns [orth~"[,-]"] ns [orth~"[0-9]+"])?
```

A used as reference in some operation will refer to a sequence of digits and B will refer to an optional hyphen with some more digits not separated by space. For numeric references, 2 will have the same effect as A, 3 will be the same as the B and 1 will refer to the optional sentence beginning from the left context.

3.4.2 Entity specifications

A single entity specification describe one syntactic entity. It can be either special entity specification, a token specification or a group specification.

Special entity specifications are sb, se and ns for matching sentence beginning, sentence end and no-space dummy entities respectively.

Token specification

A token specification can match either single segment or a syntactic word. It is a conjunction of requirements on specific attributes closed in square brackets, like this:

```
example
[pos ~~ liczba  &&  orth ~ "[12][0-9]{3}"  &&  abs(syn) !< (2+3)*-0.8]
```

The attribute name may be any of those specified in tagset or 'pos', 'base' or 'orth' (part-of-speech, lexical form and orthographic form). Value specification of any enumerable attribute can be a single word or a regular expression closed in double quotes (with standard syntax of regexps on text). The operator between them may be one of:

- ~ - there exists an interpretation which meeting the value of attribute
- ~~ - all the interpretations meet the requirement
- !~ - there does not exist any interpretation with such a value
- !~~ - not all of the interpretations meet the requirement

For numeric attributes value specification should be a correct, constant expression (see section 3.5.1 for details on expressions). Additionally, to the name of numeric attribute on the left side can also have abs() operation applied. The operator between left and right hand for "higher than" comparison can be one of:

- > - there exists an interpretation with higher value of attribute
- >> - all the interpretations must have higher value of this attribute
- !> - there does not exist any interpretation with higher value of attribute

- `!>>` - not all of the interpretations have higher values of the attribute

Similarly, `<`, `<<`, `!<`, `!<<` has corresponding meaning for ”less than” and `=`, `==`, `!=`, `!==` has for ”equality” comparison.

Group specification

A group specification uses similar syntax:

```
example
[type="Month_NG" && synh=[case~"gen"]]
```

The differences between token specification and group specification are:

- different set of attributes: `type` (a string, user defined during creation of group), `semh` (semantic head), `synh` (syntactic head) and `head` (both syntactic and semantic head; have to be the same)
- - different operators - only `=` (has to meet) and `!=` (must not meet), since groups don't have interpretations.
- - values of attributes: type is an arbitrary string, so value requirement is like in the token specification. Heads are referred to by nested token specification describing the group heads.

3.5 Operations overview

3.5.1 Common parts

The `tag_specification` used in this chapter should be in the form:

```
syntax
pos_value_spec:attr1_value_spec:attr2_value_spec and so on.
```

Every value specification of enumerable attribute can contain:

- single value for attribute or pos (e.g. subst for pos, nom for case)
- wildcards on some attributes:
  - `value1.value2.value3` gives 3 copies of the tag with each of the values
  - `attributename*` stands for all the possible values of this attribute

  Multiple wildcards multiply the number of added or set interpretations.

- references to specific enumerable attributes of matched entities: with explicitly specified entity - `1.case` or `A.gender`, or to a default entity for the operation - `case`, `gender` (in copying, 3-arg form of word or in alter it is the 1st argument of the operation)

For numeric attributes it should be an assignment of form:

syntax

```
attr_name = expression
```

where expression may be any expression (not necessary constant) using syntax defined in section

Numeric expressions

The numeric expression can use `*`, `/`, `-`, `+` binary operators, `-` and `abs()` unary operators and parentheses.

The constant expression can be built only from floating point numbers. The non-constant expression can also contain references to specific numeric attributes of matched entities, in one of two forms . A `1.sen` or `A.sen` is a reference to a value of attribute 'sen' from specific entity. A `sen` is a reference to a value of 'sen' from the default entity, which in copying, 3-arg form of word or in alter is the 1st arg of the operation.

The non-constant expression may also use values of variables (using `@variablename` notation).

3.5.2 agree

syntax

```
agree(list of attribute names, list, of, references)
```

Try to agree attributes from given list in all of the segments referenced in the coma separated list. If it cannot be done the operation fails.

example

```
agree(number gender,1,A,B);
agree(case number gender,1,2);
```

3.5.3 orthnot

syntax

```
orthnot("regular expression", reference)
```

Check if the orthographic form is matched by the regex, if so, the operation fails.

example
```
orthnot("[A-Z].*", 1);
```

3.5.4 assign

syntax
```
assign(@variablename, expression)
```

Assign a value of expression to a given variable. The expression may be not constant - it will be calculated during execution of action. It can also refer to the old value of the assigned variable.

example
```
assign(@counter, @counter+1);
assign(@total = @total + abs(@counter + B.sen)*@ratio);
```

3.5.5 unify

syntax
```
unify(list of attribute names, list, of, references)
```

This operation is like the agree, but also deletes every interpretation not agreed. Also can fail.

example
```
unify(number gender,1,A,B);
unify(case number gender,1,2);
```

3.5.6 persistent_unify

syntax
```
persistent_unify(list of attribute names, list, of, references)
```

This operation is like the unify, but makes the unification persistent - referenced segments will be reunified every time one of them will change (even if they are closed in syntactic word group and inaccessible from subsequent rules).

example
```
persistent_unify(case number gender, 1, 2, 4, 6);
```

3.5.7 add

syntax

```
add(tag_specification, "base form", reference)
```

Add interpretation(s) to the referenced token. If base form is omitted, the first interpretation's base form is used.

See section 3.5.1 for syntax of `tag_specification`.

example

```
add(adj:sg:loc:m3:pos,,1);
add(adj:pl:dat:gender*:pos,,1);
```

3.5.8 set

syntax

```
set(tag_specification, "base form", reference)
```

This operation is like add, except that it first removes all the interpretations the referenced token had before

See section 3.5.1 for syntax of `tag_specification`.

example

```
set(adj:sg:gen:m1:pos,,1);
set(num:pl:nom.acc.voc:m2.m3.n.f,"ile",A);
```

3.5.9 delete

syntax

```
delete(<interpretation requirement>, reference)
```

Delete all interpretations from referenced token matched by requirement. The requirement is basically a segment requirement without square brackets.

example

```
delete(pos~impt,2);
delete(pos~ger && case~voc, 1);
```

See section 3.4.2 for syntax of segment requirements.

3.5.10 leave

syntax

```
leave(<interpretation requirement>, reference)
```

It is an opposite to delete - deletes all interpretations that are not matched.

example

```
leave(pos~qub, 4);
leave(pos~"winien|praet|bedzie|fin|impt|imps|inf" && number~pl,C);
```

3.5.11 word

syntax

```
word(tag_specification, base_specification)
word(reference, partial_tag_specification, base_specification)
```

Build a syntactic word from whole Match section with specified interpretations. The base specification can consist of sequence of strings enclosed with double quotes and references to base or orth in the form: 1.base or B.orth All the parts of specification is concatenated to form the base. A special reference to 0 (zero) gives concatenated values (bases or orths) from the whole Match section *including* spaces (unless there exist a ns).

The 2-argument version creates interpretations like the add operation. The 3-argument version copies interpretations from the referenced segment and modifies them using the value of $partial_tag$. The tag can contain values of some attributes, or even change the POS. In case if the created tag would not conform to the tagset Spejd will warn and (optionally) try to correct the tag. Several sets of interpretations can be added by providing multiple sets of arguments separated by semicolon.

See section 3.5.1 for syntax of `tag_specification`.

example

```
word(num:pl:case*:gender*:rec, "miliard";
     subst:number*:case*:m3, "miliard");

word(subst:sg:case*:m3, "rok");
word(adj:1.number:nom:m1:pos, 1.orth "-owski");

word(2, aff, base);
word(3, neg, 0.base);
```

3.5.12 alter

syntax

```
alter(reference, partial_tag_specification, base_specification)
```

This action is similar to 3-arg word action, but doesn't build a new syntactic word. It modifies a segment or a word pointed by reference in-place, in the similar manner to the set action (replaces interpretations). The difference between alter and set is that alter uses "3-arg word"-like interpretation building mechanism, which allows to modify only some parts of tag without need to specify the whole complete tag. Unlike word, the alter action doesn't allow multiple sets of arguments.

See section 3.5.1 for syntax of `tag_specification`.

example

```
alter(3, imp:refl, base);
alter(2, inf:imp:1.neg, B.base);
```

3.5.13 group

syntax

```
group(group_type, syntactic_head_reference,
      semantic_head_reference)
group(group_type, syntactic_head_reference,
      semantic_head_reference, base_specification)
group(inherit_reference, syntactic_head_reference,
      semantic_head_reference)
group(inherit_reference, syntactic_head_reference,
      semantic_head_reference, base_specification)
```

Create a syntactic group from the whole Match section with *type* being arbitrary word, heads given by references, and optional lexical form given by specification with the same syntax as the base specification in the word operation. If the lexical form is not given it defaults to 0.orth. If a head refers to another group, it's head will be used. If a reference to a group is given instead of group type name, the new group's type will be inherited from the referenced group.

example

```
group(NG,2,3);
group(1,2,1, 0.orth);
group(A,2,2, 0.orth);

# note: A in the last example is a reference,
# not a group type (it must not be single capital letter).
```

See section 3.5.11 for syntax of `base_specification`.

3.5.14 join

syntax

```
join(group_type, syntactic_head_reference,
     semantic_head_reference)
join(group_type, syntactic_head_reference,
     semantic_head_reference, base_specification)
join(inherit_reference, syntactic_head_reference,
     semantic_head_reference)
join(inherit_reference, syntactic_head_reference,
     semantic_head_reference, base_specification)
```

Join all the entities in the Match section together forming a syntactic group. If the Match contains groups, they are destroyed and their contents are put into the group. Any token from the Match is put into the group without changes. Arguments mean the same as in the group operation.

example

```
join(NG,2,2);
join(1,2,2, base);
join(A,2,2, A.base " " B.base);

# note: A in the last example is a reference,
# not a group type (it must not be single capital letter).
```

See section 3.5.11 for syntax of `base_specification`.

3.5.15 attach

syntax

```
attach(reference)
attach(reference, base_specification)
attach(group_type, reference)
attach(group_type, reference, base_specification)
```

Attach the rest of Match to the referenced group. Optionally sets different lexical form and changes the type of resulting group. Preserves all the heads.

example

```
attach(A);
attach(ADJ_G, 3);
attach(A, 0.base);
```

See section 3.5.11 for syntax of `base_specification`.