User's guide to the SHALMANESER system

Katrin Erk and Sebastian Padó
{erk,pado}@coli.uni-sb.de

June 12, 2012

Contents

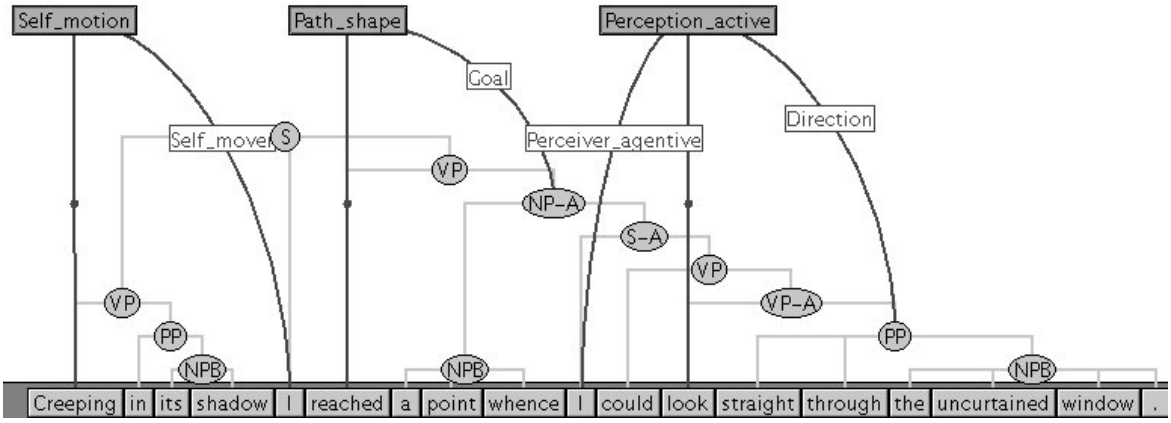

Figure 1: A sample analysis by SHALMANESER (Arthur Conan Doyle: The Hound of the Baskervilles)

1. Introduction

SHALMANESER is a supervised learning toolbox for shallow semantic parsing, i.e. the automatic assignment of semantic classes and roles to text. The system was developed for Frame Semantics; thus we use Frame Semantics terminology and call the classes frames and the roles frame elements. However, the architecture is reasonably general: It can handle any role-semantic paradigm (e.g., PropBank roles) and any set of word senses (e.g., WordNet synsets), provided the input data is offered in SalsaTigerXML.

SHALMANESER caters both for end users, and for researchers. For end users, we provide a simple end user mode which can simply apply the pre-trained classifiers for English (FrameNet annotation / Collins parser) and German (Salsa Frame annotation / Sleepy parser). For researchers interested in investigating shallow semantic parsing, our system is extensively configurable and extendable.

For a conceptual description of SHALMANESER, please consult Erk and Pado (2006), available in the present directory as `shal_lrec.pdf`. This document is the SHALMANESER user's guide. It proceeds from more general to more specific topics. This section introduces the system, its structure and its modes of operation. Section 2 describes how to use the end user mode. Section 3 treats the manual mode. Section 6 discusses some ways of extending SHALMANESER.

1.1. Shallow semantic analysis

Figure 1 shows a sample analysis by SHALMANESER. The syntactic analysis (in light grey) was done using the Collins parser. SHALMANESER has identified and tagged three predicates in this sentence: *Creeping* has been identified as having the sense SELF_MOTION (an animate being moves under its own power), and its SELF_MOVER role has been assigned to *I*. The word *reached* has been wrongly disambiguated to PATH_SHAPE (shape of a road) rather than ARRIVING, however the semantic role GOAL (*a point whence I could look straight through the uncurtained window*) is correct. The word *look* has been correctly identified as having the sense

PERCEPTION_ACTIVE (a perceiver wilfully directs his attention to some phenomenon), with semantic roles PERCEIVER_AGENTIVE and DIRECTION.

Note that each predicate with its roles is represented separately; the system does not try to construct a single connected predicate-argument structure. Note also that roles of the predicates point to nodes of the syntactic structure.

1.2. Structure

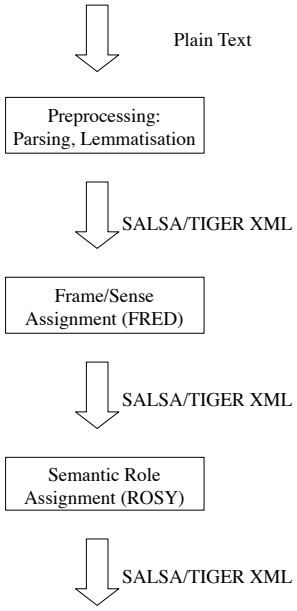

Figure 2: Architecture overview

Figure 2 shows the processing architecture of the system. It is designed as a feed-forward system, all stages of which communicate through an XML format called SalsaTigerXML (Erk and Pado 2004). This paper is available as `salsatigerxml.pdf` in the current directory.

SHALMANESER currently has three stages:

1. FRPREP: FRPREP is the preprocessor for both FRED and ROSY: It takes input as either plain text, SalsaTigerXML, or the FrameNet LU XML format. It takes care of the following steps:
   a) Re-encoding of the input (Iso-Latin-1 vs. UTF-8)
   b) Lemmatisation
   c) POS tagging
   d) Parsing
   e) Production of SalsaTigerXML.

Language	Step	Program(s)
English	POS-Tagging/ Lemmatisation	TreeTagger (Schmid 1994)
	Parsing	Collins' Parser (Collins 1997), Minipar (Lin 1993)
German	POS-Tagging	(done by Sleepy)
	Lemmatisation	TreeTagger (Schmid 1994)
	Parsing	Sleepy (Dubey 2005)

Table 1: Currently supported preprocessing programs in FRPREP

Framework	System
MBL	TiMBL (Daelemans, Zavrel, van der Sloot, and van den Bosch 2003)
Maximum Entropy	Mallet (McCallum 2002)

Table 2: Currently supported ML systems in ROSY

Lemmatisation, POS Tagging and Parsing are delegated to external programs. Table 1 shows the preprocessing programs which are currently supported.

Note that parsing is optional and not strictly necessary; if no parser is employed, the system constructs a flat tree with one nonterminal node "S" which directly dominates all terminals. This representation is also used for sentences where parsing fails; however, it is not recommended for actual role assignment, since the path from target to a constituent is usually considered one of the strongest indicators for decising on the status of a potential argument.

2. FRED: The FRED (FRamE Disambiguation) system assigns frames, modelling the task as word sense disambiguation. The system uses a Naive Bayes learner and relies mainly on bag-of-words context features (Erk 2005). This paper is available as `fred.pdf` in the current directory.

3. ROSY: The ROSY system (ROle assignment SYstem) system assigns frame-semantic roles to all predicates which have been assigned a frame by Fred or any other system. For an introduction to semantic role assignment, see e.g. (Gildea and Jurafsky 2002). Rosy delegates the actual statistical modelling to an external ML system; see Table 2 for an overview of interfaces we provide.

SHALMANESER comes with classifiers pre-trained on the English FrameNet corpus (Collins parser) and on the German Salsa corpus (Sleepy parser).

1.3. Usage

Each of the 3 systems involved, FRPREP, FRED as well as ROSY, is controlled by an **experiment file** that describes all relevant properties of the current "experiment", for example the location and format of the data, system parameters, and the features to be used in learning.

End user mode. In End user mode, default experiment files provided with SHALMANESER are used. Using the end user mode therefore requires only minimal configuration, namely a call of the **setup script** we provide, which creates customised **experiment files** for your system: it sets the paths to all external components (see Section 1.5).

From then on, end users just call the bash script `shalmaneser.sh`, which processes free text fully automatically to produce a version with frame-semantic analysis. Naturally, this ease of use comes at the expense of flexibility; in End User mode, it is not possible to (re-)train classifiers on new data.

A usage scenario for the end user mode is provided in Section 2.

Manual mode. In manual mode, users call the components FRPREP, FRED, ROSY manually and can adapt the experiment files individually. To provide an easier start, we provide **sample experiment files** which contain reasonable default values where possible.

Usage scenarios for the manual mode are described in Section 3.

Input format. If your data is already in SalsaTigerXML, we recommend to run it through FRPREP anyway to normalise it. If you use plain text, it has to be tokenised, one sentence per line.

1.4. Prerequisites

You need the following items installed on your system:

- Ruby, at least version 1.6.
- A MySQL database server. The server must be large enough to hold the test data (in end user mode) plus any training data (for training new models in manual mode). For example, training on the complete FrameNet 1.2 dataset requires about 1.5 GB space.
- The **MySQL/Ruby** package (`http://www.tmtm.org/en/mysql/ruby/`), at least version 2.6. Earlier version might work, but have been found to act strangely.
- If you don't want to train classifiers from you own data, you need to download suitable classifiers from our homepage (see Table 3) for available configurations.
- Preprocessing tools for your language, at least the ones required for the use the pre-trained classifiers. Currently SHALMANESER provides interfaces for the following systems:

System	*Version*
TreeTagger	(README from 09.04.96)
Collins	1.0
Minipar	(README from 1998)
Sleepy	`sleepy3`

- At least one machine learning system for ROSY. Currently SHALMANESER provides interfaces for the following systems:

System	*Version*
TiMBL	Timbl5
Mallet	mallet 0.4

Note: Please make sure you run the system in a terminal with Unicode encoding (`export LANG=eng_US.UTF-8`).

1.5. Setting up SHALMANESER on your system

There is a setup script, (call `ruby setup.rb`), which will ask you for information on your system, especially paths to preprocessing software. It will produce suitable experiment files for you to directly enter the Enduser mode (see Section 2).

We strongly recommend running `setup.rb` even if you only want to run the system in manual mode, for a number of reasons. The most important one is that a number of programs, specifically `treetagger` and `mallet`, require special interfaces to cooperate with SHALMANESER. We have provided these interfaces, but they need to be initialised, which is what happens in `setup.rb`.

In addition, `mallet`, versions 0.3.2 and 0.4 alike, exhibits a behaviour we consider a bug, namely that it cannot deal with class labels in the test data unseen in the training data. This leads to occasional exceptions when classifiers are applied. The batch file `program/tools/mallet-patch.sh` can install a workaround, and can be called automatically during setup. Note: You need write permissions on the `mallet` directory to successfully install the patch. If you do not, please ask your local superuser to execute `program/tools/mallet-patch.sh`.

1.6. Data Visualisation

The output format of SHALMANESER, SalsaTigerXML, is exactly the input format of SALTO, a graphical user interface for semantic role annotation. SALTO thus offers itself for simple data inspection (and possible manual post-processing). The sample annotation in Figure 1 is a snapshot from SALTO.

SALTO can be obtained free of charge from `http://www.coli.uni-saarland.de/projects/salsa/page.php?id=software`. Currently, the best way to load a processed corpus into SALTO is to use the menu item File → Open File[1]. SALTO can also be called directly from within SHALMANESER with the `-v` option in enduser mode (see the next section).

[1] Suggestion: Unchecking the checkbox "Use native file chooser" in the Edit → Preferences dialogue can lead to a more pleasant browsing experience.

Language	Step	Program(s)
English	POS-Tagging/ Lemmatisation	TreeTagger (Schmid 1994)
	Parsing	Collins' Parser (Collins 1997)
	Classifier	Mallet (McCallum 2002)
German	POS-Tagging	(done by Sleepy)
	Lemmatisation	TreeTagger (Schmid 1994)
	Parsing	Sleepy (Dubey 2005)
	Classifier	Mallet (McCallum 2002)

Table 3: Settings for the pre-trained classifiers in Enduser mode.

2. Using SHALMANESER in Enduser mode

To use SHALMANESER in end user mode, set up the system once (Section 1.5) and then just call the bash script `shalmaneser.sh`. `shalmaneser.sh -help` lists the possible command line parameters:

```
shalmaneser.sh -i inputpath [-o outputpath -e encoding
                             -l language -p parser]

--input          -i      Input path
                         Location of the input files
--output         -o      Output path
                         Path for output (annotated files)
                         Default: $HOME
--encoding       -e      Encoding
                         Encoding format of the files
                         iso / utf8 / hex (Default: iso)
--language       -l      Language of the input files
                         de / en (Default: de)
--parser         -p      Choice of parser
                         sleepy / collins / minipar (Default: sleepy)
--visualize      -v      Display results directly with SALTO
                         (presuming it is installed)
```

2.1. Usage scenario: Using the pre-trained classifier

- **First time setup:** Download pre-trained classifiers (see Table 3)
- **First time setup:** Run `setup.rb` to create appropriate experiment files (preprocessing settings must match pre-trained classifiers!)
- Run `shalmaneser.sh`. For English, a sample call could be

```
bash shalmaneser.sh -i ~/data-for-analysis -o ~/analysed-data
                    -l en -p collins
```

A corresponding call for German:

```
bash shalmaneser.sh -i ~/data-for-analysis -o ~/analysed-data
                    -l de -p sleepy
```

- Everything (preprocessing, frame and role assignment) will happen fully automatically.

3. Using SHALMANESER in Manual mode: Usage Scenarios

It is necessary to use SHALMANESER in manual mode if using the pre-trained classifiers is not enough for you, e.g. if you want to train your own classifiers, add new features (which entails re-training as well), etc.

`setup.sh` instantiates the general (system-wide) settings and saves them in the directory `SampleExperimentFiles`. There is one sample experiment file for each combination of a module (FRPREP, FRED, and ROSY), and a parser (Choice of the parser specifies the language as well). In Manual mode, the experiment-specific entries must be added manually – we recommend copying the sample files somewhere else to do that. In the sample files, the options which need filling in are marked by percent signs (%).

In this section, we first describe how to train classifiers, and then how to apply them to new data (the second step is the manual version of what the Enduser mode does).

At the end of the document, we show instantiated experiment files that are referred to in the following text. These example files omit some comments and some more obscure settings; these are all explained in the sample experiment files (`program/SampleExperimentFiles`).

We also recommend browsing the detailed command line help texts provided by all three programs:

```
ruby frprep.rb --help
ruby fred.rb --help
ruby rosy.rb --help
```

3.1. Usage scenario 1: Training classifiers on German data

In this scenario, we train the frame and role assignment systems on new data. We assume that we have German data that is annotated with frames and semantic roles and that is represented in SalsaTigerXML (e.g. data annotated by the Salsa project, or any German data annotated manually using SALTO). Training proceeds in the order of the processing chain: first preprocessing, then frame assignment, then role assignment.

Preprocessing. The instantiated experiment file for preprocessing is provided in Fig. 10. First the **data paths** have to be set:

`directory_input`**:** We assume that `/home/marie/all_salsa_data/` holds the annotated input data

`directory_preprocessed`**:** We assume that the preprocessed data can go to `/home/marie/all_salsa_data_preprocessed`.

If a directory does not exist, FRPREP will create the directory for you.

Next, the **data description** in `language`, `origin`, `format` and `encoding` reflects the facts that the input data is German and stems from Salsa annotation, that it is in SalsaTigerXML and that it is encoded in ISO-8859-1.

FRPREP offers different **preprocessing steps**: lemmatisation, part-of-speech tagging and parsing; each of these steps can be turned on or off. Which steps make sense in the present scenario? We assume that the German training data is already manually syntactically annotated; however, you will probably want to re-parse the data with Sleepy. The reason for this is that you probably want to use the trained system to analyze new plaintext data, which has to be parsed automatically as well. Since automatic parsing usually introduces errors, it makes sense to expose the system during training to the kind of errors it will encounter later on. Hence you set `do_parse = true`. Furthermore you set `do_lemmatize = true` since the TIGER corpus does not include lemma information at the moment. The Sleepy parser adds part-of-speech tags anyway, so the POS tagging parameters are not set.

The remaining parameters describe the **unique experiment ID tag** applied to the data for this system run `frprep_experiment_ID`), the location where FRPREP stores temporary data (`frprep_directory`), and the location and parameters of the syntactic processing systems that FRPREP uses.

Provided the instantiated experiment file is called `exp_preproc.salsa`, you can finally **start preprocessing** with:

```
ruby frprep.rb -e exp_preproc.salsa
```

Frame assignment. Next, you can train the frame assignment system on the preprocessed Salsa data. An appropriate experiment file for FRED is provided in Fig. 11 and 12.

Again, we have to set the **experiment ID tag**, `experiment_ID`. FRED will use this tag to mark the data for this system run. Then we specify the **data paths**:

`directory_output`**:** This specifies the location of FRED's output. As example, we have set it to `/home/marie/fred_output/`. However, since we are in the training phase right now, no output will be produced.

`preproc_descr_file_train`**:** This is the location of the *experiment file* used by FRPREP to process the *training* data. FRED needs access to this file to know about the parser etc. It also infers the location of its input data (the output location of FRPREP) from this parameter[2].

[2]There is a corresponding parameter (`preproc_descr_file_test`) for *testing* data as well, but it does not need to be set during the training phase.

`classifier_dir`**:** This is where the classifiers are written during training. If the parameter is not specified, classifiers are written by default to a subdirectory of `data_dir`.

Since we are instantiating the experiment file ourselves this time, we set the **enduser mode** (`enduser_mode`) to `false`. With respect to actual **classification**, the example experiment file contains the following relevant parameters:

`classifier` is set to Naive Bayes. This is currently the only implemented classification paradigm for FRED– see the file `fred.pdf` in this directory for details,

`smoothing_lambda` is a smoothing factor of 0.55 to deal with unseen data; it is set to 0.55[3].

`features` is a list of features to be used for classification. The Naive Bayes classifier currently supports bag-of-word context (`context`), 2- and 3-grams of words surrounding the target (`ngram`), headwords of grammatical functions of the target (`grfunc`) and the grammatical functions themselves (`syn`). A feature weight is given to each feature[3].

`feature_dim` The context and n-gram features can be specified over word forms, lemmas, or parts of speech. This feature is a list of the dimensions that should be used.

`window_size` If the TIGER corpus is used, FRED can use words from surrounding sentences as context as well; this is indicated by setting this feature to the number of adjacent sentences to consider. This requires pointing the parameter `tigerfile` to the location of the TIGER corpus (as XML).

There is one more relevant setting: the directory where FRED will put its internal data (contents of this directory are deleted at the beginning of each FRED run) (`fred_directory`). This should usually be set to a temporary directory. The remaining settings are FRED-internal ones, and can be left unchanged.

To **start training**, you first transform each input instance to a feature vector and then you train FRED on this featurized data:

```
ruby fred.rb -t featurize -e exp_fred.salsa -d train
ruby fred.rb -t train -e exp_fred.salsa
```

Recall that the classifiers will be written to `classifier_dir`.

Role assignment. Lastly, the role assignment system ROSY has to be trained on the corpus. Again, we have prepared an appropriate experiment file (see Fig. 13 and 14).

After enduser mode and experiment id are specified in the same way as for FRED, it is time for the **data paths**:

`directory_output`**:** This parameter specifies the location for ROSY's output (same as for FRED). However, again, no output will be produced (training phase).

`preproc_descr_file_train`**:** This is the location of the *experiment file* used by FRPREP to process the *training* data (same as for FRED).

[3]The experiment files in `program/SampleExperimentFiles` list recommended settings for different parsers.

`directory_input_train`**:** This parameter specifies where ROSY looks for its input data. Unlike FRED, ROSY does not infer this parameter from the FRPREP experiment file. The reason is that during classifier *application*, the pre-processed data is first run through FRED for frame assignment, and then run through ROSY for role assignment; therefore, ROSY usually does not always read FRPREP's output.

`classifier_dir`**:** This is where the classifiers are written during training. If the parameter is not specified, classifiers are written by default to a subdirectory of `data_dir`.

The next block of parameters concerns **data correction**: ROSY integrates two mechanisms, which are explained in more detail in Pado and Erk (2006). The first corrects obvious syntactic inconsistencies in the input data, both with respect to the span of FEs (`fe_syn_repair`) and relative clauses (`fe_rel_repair`). For German, we recommend to use only `syn_repair`; for English, both can be used.

The second mechanism is a pruning mechanism which deletes all nodes of the syntactic structure that are rather improbable as frame elements, implementing the heuristic developed by Xue and Palmer (2004). The parameter takes a feature name as value, and removes all nodes for which the value of this feature is `false`. We have implemented a feature called `prune` which works for both languages[4]

Next, we set the **classification settings**: The first choice concerns the "data unit" for the different processing steps (for argument recognition: `xwise_argrec`, for argument labelling: `xwise_arglab`, or for combined processing: `xwise_onestep`). The standard setting of these parameters is `frame`, i.e. one classifier per frame; other possibilities are given in the comments of the sample files. `assume_argrec_perfect` is a special setting which can be used to investigate argument labelling only (e.g. as in Erk and Pado (2005)) by assuming perfect argument recognition (argument boundary detection). We have deactivated it here.

Next, the `classifier` and its path are specified, and the `features` it can use. Features are available language-independently, however it depends on the chosen parser whether particular features contain sensible values, or just some default. Section <LALA> contains a table with all currently implemented features and lists their semantics.

As with FRPREP and FRED, you finally specify a directory for the system's internal data, `data_dir`. The actual location of your experiment's data is a subdirectory of `data_dir` named after your experiment ID.

To **start training**, you first transform your input data to feature vectors, and then train ROSY on this featurized data (same as for FRED). The command `-d` specifies that we are processing *training* data, which will be written in a special table in the database called *train* which will be used for any training steps.

ROSY can split role assignment into two sequential classification steps (recommended): The first one distinguishes roles from non-roles (argument recognition), the second one gives roles their labels (argument labelling). You have to train the argrec classifier first, since training the arglab classifier requires a working argrec classifier.

```
ruby rosy.rb -t featurize -e exp_rosy.salsa -d train
```

[4]Note that the pre-trained classifier for German does not perform pruning; the pruning feature is thus not computed.

```
ruby rosy.rb -t train -e exp_rosy.salsa -s argrec
ruby rosy.rb -t train -e exp_rosy.salsa -s arglab
```

It is possible to train a combined classifier for argrec and arglab by stating instead:

```
ruby rosy.rb -t featurize -e exp_rosy.salsa -d train
ruby rosy.rb -t train -e exp_rosy.salsa -s onestep
```

Again, classifiers will be written to `classifier_dir`.

3.2. Usage scenario 2: Training from FrameNet data

Training on English data with FrameNet annotation can follow, in principle, the directions given for Usage scenario 1. In the following, we only list the parameter settings that have to be changed accordingly. We assume that Collins' parser is used.

Preprocessing. Both the data description and the preprocessing parameters have to be adapted. For English, we perform separate POS tagging using Treetagger and use Collins' parser for syntactic analysis. TreeTagger does not need to be adapted.

```
language = en
format = FNXml
origin = FrameNet

pos_tagger = treetagger
pos_tagger_path = /usr/local/Shalmaneser1.1/program/tools/treetagger/shal-eng

parser = collins
parser_path = /proj/software/collins/
parser_max_sent_num = 2000
parser_max_sent_len = 80
```

The format option `FNXml` indicates the XML format used for the FrameNet lexical unit (LU) files. To process the FrameNet full corpus annotation, which has a different XML format, put

```
format = FNCorpusXML
```

instead.

Frame assignment. For frame assignment, we only change the feature weights. (It is also reasonable to change `classifier_dir`, unless you want to have your previously trained classifiers overwritten.)

```
feature = context 1.0 0.85
feature = ngram 1.6
feature = grfunc 1.1
feature = syn 4.0
feature = synsem 2.2
```

Role assignment. For English data, we can take advantage of both data correction mechanisms:

```
fe_syn_repair = true
fe_rel_repair = true
```

Again, you will also want to change `classifier_dir`.

3.3. Usage Scenario 3: Analysing new German text

This section treats the task of semantically analysing new text using FRED and ROSY. In this scenario, we assume that this is German plaintext, i.e. just words without any linguistic annotation. Semantically analysing such text presupposes that you either have a set of pre-trained classifiers, or that you have trained classifiers according to the instructions in Section 3.1. (The same scheme can be applied to new English text, using the instructions in Section 3.2).

As before, you have to follow the three steps of preprocessing, frame assignment using FRED, and role assignment using ROSY (cf. Figure 2). We recommend to re-use the experiment files that you used for training the classifiers. Especially the classification and data correction settings need to agree between training and testing data; e.g., the test data needs to use the same feature set as the classifier. What you will need to adapt are the data description and data path settings. The main difference to training is that you will be calling FRED and ROSY in *testing* mode instead of training mode.

Preprocessing. We assume that we write a new experiment file for preprocessing, called `exp_preproc.mytest`. The main difference to `exp_preproc.salsa` as described in Section 3.1 concerns the *data description* section:

```
language = de
format   = plain
encoding = iso
```

We also specify new directories for input and output data:

```
directory_input = /home/marie/plaintext/
directory_preprocessed = /home/marie/plaintext_preprocessed/
```

The input data `format` is `plain`, and the `origin` is not specified at all, since this is just arbitrary text with no special annotation. We retain all preprocessing steps.

FRPREP can can then be called with:

```
ruby frprep.rb -e exp_preproc.mytext
```

Frame assignment. For testing, we call FRED's `featurize` task with the option `-d test`, thereby running it on the test dataset. By specifying `-d test`, FRED reads the preprocessing settings from the experiment file specified as the value `preproc_desc_file_test`. In the present case, the test data settings can be read from `exp_preproc.mytext` (Figure 11). The corresponding file for training data, `preproc_desc_file_train`, if it exists, is ignored during test runs.

```
ruby fred.rb -t featurize -e exp_fred.salsa -d test
```

After featurization, we can run FRED's `test` module. The test module automatically applies the classifiers it has trained during the training session (see again Section 3.1) on the test data:

```
ruby fred.rb -t test -e exp_fred.salsa
```

During testing, the `directory_output` parameter is actually used by FRED to store its output as SalsaTigerXML files, which can already be inspected in SALTO too.

Role assignment. Recall that ROSY saves its data in database tables, and that training data is saved in a special table called *train*. Since you might want to apply the classifier to different datasets, you have to specify a *test table id* with `-i`. This id is used by ROSY to create (or overwrite) a database table with the new information. Here, we use the test table id `plaintest`.

Again, featurization of the test data is the first step. We re-use the experiment file `exp_rosy.salsa`. As for frame assignment, specifying `-d test` lets the featurization step import the data properties from the experiment file specified for the test data (`preproc_desc_file_test`). After featurisation, we can now apply (by specifying `-t test`) the two classifiers (for argument recognition and argument labelling) we trained in Section 3.1 to our plaintext data, in the old order: first argrec, then arglab. As for featurization, the parameter `-i plaintest` specifies that the test data is stored in the "plaintest" table. ROSY assigns roles to all targets with a frame (i.e. in our scenario, all known targets). The output of the test is written to the directory given by the parameter `directory_output`.

```
ruby rosy.rb -t featurize -e exp_rosy.salsa -d test -i plaintest
ruby rosy.rb -t test -e exp_rosy.salsa -s argrec    -i plaintest
ruby rosy.rb -t test -e exp_rosy.salsa -s arglab    -i plaintest
```

4. Using SHALMANESER in manual mode: Additional features

This section lists shortly some additional features that FRED and ROSY provide. For details of the command line calls see the command line help pages:

```
ruby fred.rb --help
ruby rosy.rb --help
```

4.1. FRED: parameter exploration mode

FRED has a special mode, `-t parameters`, to optimise the parameters for any given dataset.

4.2. FRED, ROSY: Splitting data

Both FRED and ROSY have special modes to split data into training and test sets. They can be called with the `-t split` command line option. The data is split sentence-wise, and the (internal) sentence IDs of the sentences are stored in a file and can be re-used, for example to evaluate different classification settings on the same split.

The following example uses FRED to split the data in an 80/20 ratio and stores information about sentence IDs under the ID `myLog`.

```
ruby fred.rb -t split -e fred.expfile --logID myLog --trainpercent 80
```

4.3. FRED, ROSY: Evaluation

FRED and ROSY have built-in evaluation modules (`-t eval`), which provide either F-Score (Precision, Recall) or Accuracy numbers for any testing run.

4.4. ROSY: Database administration

As mentioned in Section 3.3, ROSY uses a database to store train and test data. Of course, this database needs to be accessible for inspection and administration. This is possible using ROSY's `-t inspect` and `-t services` parameters. The `inspect` tasks can list all tables, the content of the tables, and the classification runs of the current experiment. The `services` task allows the user to dump the content of tables into a file (and to read dumped content from files), and to delete tables, runs and experiments.

4.5. Feature file extraction

While SHALMANESER includes its own classifiers, it may still be interesting to access the feature sets it constructs, for example in order to explore and classify the data with an external system.

If FRED has been used in non-enduser mode (in enduser mode, all internal data, including the feature files, is deleted at the end of the run), the feature files can be found in the directory `<fred_directory>/<experiment_ID>/<dataset>/features`, where `<fred_directory>` is the FRED internal directory set in the experiment file, `<experiment_ID>` is the experiment

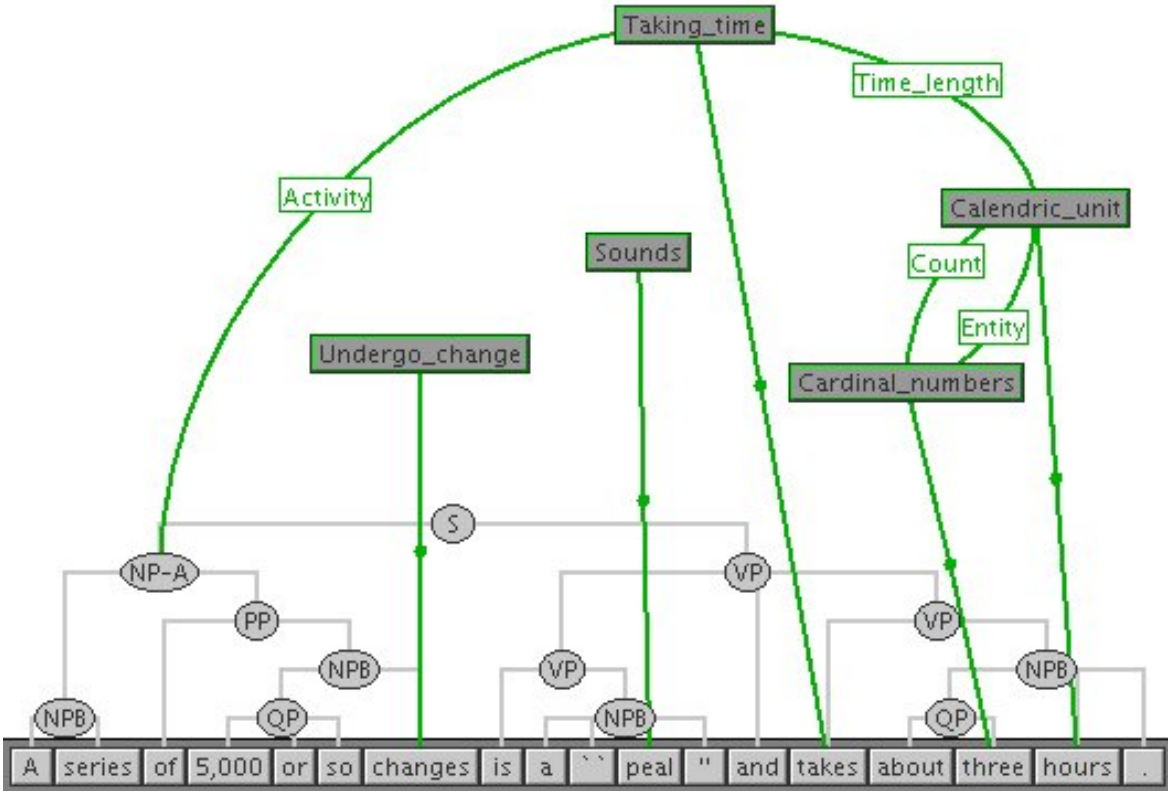

Figure 3: Deeper frame annotation: sample output

ID set in the experiment file, and `<dataset>` is `train` for training data and `test` for test data.

ROSY keeps its feature sets in the database. They can be written to files using `-t services` parameter of ROSY together with the option `-writefeatures`.

4.6. "Deeper" frame annotation

In the semantic analysis that SHALMANESER provides, all semantic roles point to constituents of the syntactic structure. An alternative representation is to have a *deep* frame structure, where a semantic role may point to either another frame or to a constituent of the syntactic structure.

SHALMANESER now comes with a tool that will "deepen" a flat semantic analysis using the following rules:

- If a semantic role r points to a syntactic constituent whose head evokes a frame f, then make r point to f instead.
- If a semantic role r points to a constituent that is a PP or an SBAR, and the head of the *complement* of that constituent (in the case of a PP, an NP; in the case of an SBAR, a VP) evokes a frame f, then make r point to f instead.

The tool is `deepen_frame_anno.rb` and is used as follows:

```
ruby deepen_frame_anno.rb <exp> <indir> <outdir>
```

where <exp> is the name of the frprep experiment file used to produce the data in the input directory <indir>, and the tool processes the data in <indir> and stores the results in files of the same name in <outdir>.

SALTO can now represent deeper frame structure. Figure 3 shows a sample output of deepen_frame_anno, applied to the manually semantically annotated corpus PropBankCorpus.xml that comes with FrameNet 1.3.

5. Implemented features

5.1. Implemented features: FRED

All word-related features exist in four variations (*feature dimensions*):

the word itself:	`feature_dim = word`
its lemma:	`feature_dim = lemma`
its part of speech:	`feature_dim = pos`
and its named entity:	`feature_dim = ne`

(The named entity features can be used only when a NE tagger has been used during preprocessing; this feature is not yet integrated into FRPREP). All FRED features are weighted. The following weights are set in the experiment file along with the feature types themselves:

- a general feature weight that applies uniformly for all features of the given type;
- distance factor: The weight of each context feature is multiplied by ⟨distance from the target word, in words⟩ * ⟨distance factor⟩.

The first column specifies the name of a feature. The next two columns indicates whether the four feature dimensions above apply to the feature, and which weights have to be be set for it (wt = feature weight; dwt = distance factor). The last column contains a short description.

Name	Dim?	Param	Description
context	y	wt, dwt	Bag-of-words context, by default comprising the current sentence. The number of context sentences can be increased by setting the parameter `window_size` to a value > 0.
ngram	y	wt	2-grams and 3-grams of words centered around the target.
grfunc	y	wt	Headwords of the dependents of the target word.
syn	n	wt	This feature group comprises three features: (1) for each dependent of the target word, the function name; (2) the concatenation of the function names of all the target's dependents; (3) for verbal targets, the voice.
synsem	y	wt	This feature combines syn and grfunc: for each dependent of the target word, the function name and the headword are concatenated.
fe	y	wt	Headwords of the semantic roles of the target word. (This feature is available only when semantic role analyses are already present in the data.)

5.2. Implemented features: ROSY

The first column specifies the name of a feature. The next two columns state whether it is available for English (i.e., (a) implemented to give reasonable results, and (b) used in the pre-trained classifier). The next two columns answer the same questions for German. The last column contains a short description.

Name	English		German		Description
	Avail	Used	Avail	Used	
pt_path	x	x	x	x	Path between target and current node, as phrase type sequence
gf_path			x	x	Path between target and current node, as grammatical function sequence
path			x	x	Path between target and current node, as sequence of (PT,GF) pairs
path_length	x	x	x	x	Length of paths between target and current node (in nodes)
pt_combined_path	x	x	x	x	pt_path plus target POS and target voice (for verbs)
gf_combined_path			x	x	gf_path plus target POS and target voice (for verbs)
combined_path			x	x	path plus target POS and target voice (for verbs)
pt_partial_path	x	x	x	x	Path from target to lowest common ancestor (lca) of current node and target, as phrase type sequence
gf_partial_path			x	x	Path from target to lowest common ancestor (lca) of current node and target, as grammatical function sequence
partial_path			x	x	Path from target to lowest common ancestor (lca) of current node and target, as sequence of (PT,GF) pairs
pt_gvpath	x	x	x	x	Path from verb governing the target to currnet node, as phrase type sequence
gf_gvpath			x	x	Path from verb governing the target to currnet node, as grammatical function sequence
gvpath			x	x	Path from verb governing the target to currnet node, as sequence of (PT,GF) pairs
ancestor_rule	x	x	x	x	Grammar rule expanding lca
frame	x	x	x	x	Frame (semantic class) of the target
target	x	x	x	x	Target lemma
target_pos	x	x	x	x	Target part of speech

target_voice	x	x	x	x	Target voice (for verbal targets)
gov_verb	x	x	x	x	Verb governing the target
pt	x	x	x	x	Phrase type of the current node
gf			x	x	Grammatical function the current node fills for the target (heuristically)
father_pt	x	x	x	x	Phrase type of the father node of the current node
relpos	x	x	x	x	Relative position of current node to target: left, right, including
prep	x	x	x	x	Preposition of the current node (if any)
const_head	x	x	x	x	Head lemma of current node
const_head_pos	x	x	x	x	Part of speech of current node
icont_word	x	x	x	x	Informational content word: for PPs, head of embedded NP; for S and VP: top content verb
firstword	x	x	x	x	First word in yield of current node
lastword	x	x	x	x	Last word in yield of current node
leftsib	x	x	x	x	Left sibling node of current node
rightsib	x	x	x	x	Right sibling node of current node
worddistance	x	x	x	x	Minimal distance between current node and target, measured in words
ismaxproj	x	x	x	x	Is current node a maximal projection? (binary)
nearest_node	x	x	x	x	Is current node nearest node to target according to some criterion?
prune	x	x	x		Auxiliary feature, necessary for pruning

```
################
# phrase type of the instance node
class PhraseTypeFeature < RosySingleFeatureExtractor
  PhraseTypeFeature.announce_me()

  def PhraseTypeFeature.feature_name()
    return "pt"
  end
  def PhraseTypeFeature.sql_type()
    return "VARCHAR(15)"
  end
  def PhraseTypeFeature.feature_type()
    return "syn"
  end

  ################
  private

  def compute_feature_instanceOK()
    return @@interpreter_class.simplified_pt(@@node)
  end
end
```

Figure 4: An example ROSY feature

6. Extending SHALMANESER

If you want to add or change features of either FRED or ROSY, or add support for another parser (maybe even a parser for another language), or add support for another machine learning system, you will have to extend the SHALMANESER system (although we have tried to make this as painless as possible by providing interface classes). This section describes what you need to do.

6.1. Adding or changing features for ROSY

Figure 4 shows an example of a ROSY feature. Each ROSY feature is a class that inherits from `RosySingleFeatureExtractor`. (This is for the simplest case of defining just a single feature, not a list of features). The first thing a feature class has to do in its class body is to announce its presence to the system by calling the (inherited) `announce_me()` method. It describes its service through 3 methods:

- The method `feature_name()` returns the name by which the feature can be chosen in the experiment file.

SQL type	Parameter	Description
VARCHAR(X)	X=length	string of maximum length X
CHAR(X)	X=length	string of exact length X
INT		integer
TINYINT		very small integer of unsigned range 0 to 255 or signed range -127 to 128

Table 6: SQL types for ROSY features

syn	feature based on syntactic information
sem	feature based on semantic information
ubiq	feature needed with syntactic as well as semantic feature sets
gold	(gold standard) role label
admin	administrative feature needed during feature computation. SHALMANESER does not pass any admin features to the learner.

Table 7: ROSY feature kinds

- the method `sql_type()` describes the data type of the feature. This is the type of the SQL table column in which the feature will be stored. In the feature of Figure 4 this is a string of maximum length 15 characters. The list of SQL types that SHALMANESER can handle at the moment is given in Table 6.
- The method `feature_type()` describes the type of the feature. Possible values are shown in Table 7. The feature types *gold* and *admin* are necessary to single out the true role label and adminstrative features not destined for the learner. The other feature types are currently not used; they are intended for an extension of ROSY that will allow the user to activate or deactivate all syntactic or all semantic features at once.

The actual computation of the feature is done by the private method `compute_feature_instanceOK()`, which returns the feature. (The return value must match the SQL value given in `sql_type()`). The method can access many class variables defined in `RosySingleFeatureExtractor`, in our example `@@node` and `@@interpreter_class`. These variables are computed only once for all features that are being computed for the same constituent.

Here is a complete list of these class variables: first the variable name and its type, then a short description. ROSY feature extractors use the `SalsaTigerSentence`, `SynNode` and `FrameNode` classes defined in `SalsaTigerRegXML.rb`. See Appendix B for a detailed description of the `SalsaTigerRegXML` API.

`@@node`: `SynNode` object.

The current semantic role candidate to be labeled.

`@@target`: `SynNode` object.

The (main) node representing the predicate for which semantic roles are being assigned.

@@target_pos: String.

The part of speech of the predicate.

@@target_voice: String.

The voice of the predicate: "active", "passive", or nil if not a verb.

@@frame: `FrameNode` object.

The frame of the current target predicate.

@@target_gfs: Array: String $\times$ `SynNode` object.

A list of all dependents of the target, represented as pairs of function label plus dependent node.

@@paths: Hash: String $\rightarrow$ `Path` object.

Path from `@@target` to each node of the sentence, represented as a mapping from the node ID to a `Path` object describing the path. The `Path` class is described in `AbstractSynInterface.rb`.

@@relpos: String.

Position of `@@node` relative to `@ttarget`: "LEFT", "RIGHT", or "DOM".

@@node_leftmost_terminal: `SynNode` object.

The first terminal in the yield of `@@node`.

@@node_rightmost_terminal: `SynNode` object.

The last terminal in the yield of `@@node`.

@@governing_verb: `SynNode` object.

Closest governing verb of `@@target`.

@@sent: `SalsaTigerSentence` object.

The current sentence.

@@terminals_ordered: Hash: `SynNode` object $\rightarrow$ Integer.

Maps each terminal of `@@sent` to its word index (starting with 1).

@@interpreter_class: `SynInterpreter` class.

The interpreter class for the syntactic preprocessors used for the current dataset.

@@instance_ok: Boolean.

True by default, is set to false if some grave error occurs that makes the computation of features for the current `@@node` impossible.

To make your new features visible to the SHALMANESER system, you need to include the file. Supposing your new feature resides in `MyFeatureFile.rb`, add the following line to `InputData.rb` (in `Shalmaneser1.1/program/Pkg`):

```
require "MyFeatureFile"
```

This line should go in the same place as the other feature file inclusions, e.g. right after `require "RosyFeatureExtractors"`.

To use your new feature, it suffices to list it along with the other features in your ROSY experiment file. Supposing your new feature is called `my-feature` (this is the return value of the method `feature_name()` in your class), you choose it by including the line

```
feature = my_feature
```

in the experiment file.

Features for FRED are currently not modularized to the same extent as those for ROSY, so changes or additions to FRED features will require changes to the file `FredFeaturize.rb` in the directory `Shalmaneser1.1/program`. If you would like to experiment with new FRED features, drop us a note and we'll see what we can do about further modularizing the interface.

6.2. Adding a new parser, lemmatizer or POS-tagger

There are two types classes that are relevant for adding a new syntactic preprocessing system (a parser, lemmatizer or POS-tagger). One is the *interface* class: it contains methods for running the system and – in the case of a parser – for converting the system output to SalsaTigerXML. The other is the *interpreter* class: Different systems use different labels for POS tags and constituents, so feature generation needs a way of interpreting the system output, such that it knows e.g. what an auxiliary will look like or how to identify dependents in a given system.

The interface class. All interface classes inherit from the `SynInterface` class in `AbstractSynInterface.rb`, a file located in the `Shalmaneser1.1/program/Pkg` directory. For examples of existing interfaces, see `CollinsInterface.rb` (an interface to the Collins parser) or `TreetaggerInterface.rb`, located in the same directory.

Figure 5 shows the shape of a generic interface class for a lemmatizer: All interface classes (whether for a lemmatizer, POS-tagger or parser) must announce their existence to the system via the `announce_me()` method. They describe themselves via the `system()` method, which returns name by which the system can be chosen in an experiment file, and the `service()` method, which announces the service they provide: `"lemmatizer"` or `"pos_tagger"` or `"parser"`. They then have to provide a `process_file` method, which implements the call to the system. `process_file` is supposed to run the system on the input data and write the output to the output file.

`process_file` can the following input format: The file with name `infilename` contains the input data, one word per line. Sentence boundaries are indicated by an empty line.

For lemmatizers and POS-taggers, the output format is one label per line, matching the input format. In the example lemmatizer interface class in Figure 5, this is accomplished using

```
class MyLemmatizerInterface < SynInterfaceTab
  MyLemmatizerInterface.announce_me()

  def MyLemmatizerInterface.system()
    return "my-lemmatizer"
  end

  def MyLemmatizerInterface.service()
    return "lemmatizer"
  end

  def process_file(infilename, outfilename)
    # apply my-lemmatizer to infilename,
    # write result to outfilename
    %x{my-lemmatizer #{infilename} > #{outfilename}}
  end
end
```

Figure 5: A generic interface class for a lemmatizer

`%x{...}`, which executes the statement within the curly braces in a subshell. Ruby replaces each occurrence of `#{variable}` in a string by the current value of `variable`. The call of `%x{...}` in the example assumes that `my-lemmatizer` takes one file as input, conveniently in the format that `frprep` uses, and writes its output to stdout, also in the right format.

For parsers, the output format is SalsaTigerXML, hence they have to offer one additional method, as shown in Figure 6. This method takes an input file in the parser output format and transforms it to SalsaTigerXML.

To make an interface class visible to the SHALMANESER system, all that remains is to include your class at the appropriate point: Assuming that you have placed your interface class in a new file named `MyLemmatizerInterface.rb`, put the line

```
require "MyLemmatizerInterface"
```

into the file `SynInterfaces.rb` (in `Shalmaneser1.1/program/Pkg`), right after the other interfaces (search for `require "TreetaggerInterface"`).

To use the new lemmatizer/parser/POS-tagger, it now suffices to choose it in your FRPREP experiment file. Suppose the new system is a lemmatizer called `my-lemmatizer` (this is the return value of the method `system()` in your class `MyLemmatizerInterface`), you choose it by including the lines

```
lemmatizer = my_lemmatizer
lemmatizer_path = ⟨some_path⟩
```

in the FRPREP experiment file, where ⟨some_path⟩ is the directory where `my-lemmatizer` resides. The value of `lemmatizer_path` is available within your `MyLemmatizerInterface`

class as the class variable `@program_path` (a string). If the `MyLemmatizerInterface` class needs further parameters along with the program directory, list them after the path in the `lemmatizer_path = ...` line; `@program_path` will then contain the path plus the parameters, exactly as they are specified in the experiment file.

The interpreter class. While the interface class refers to one single system, an interpreter class refers to a group of systems: maximally, this is one parser plus one lemmatizer plus one POS-tagger; it may be only part of this, e.g. if your parser does POS tags too. Examples of interpreter classes are in `CollinsInterface.rb`, `MiniparInterface.rb` and `SleepyInterface` (which inherits almost all of its functionality from a class in `Tiger.rb`), all located in `Shalmaneser1.1/program/Pkg`.

Figure 7 shows a minimal interpreter class. Like an interface class it announces itself, and it describes the services offered by the class, both obligatory and optional. The figure shows a minimal interpreter class. For all other methods, the `SynInterpreter` class in `Abstract-SynInterface.rb` (`Shalmaneser1.1/program/Pkg`) offers defaults that your interpreter class may but need not overwrite. Note that the methods of the `SynInterpreter` class make heavy use of the `SalsaTigerSentence` class, which offers graph-based access to SalsaTigerXML sentences and which is described in Appendix B.

An interpreter class needs to be made visible to SHALMANESER in exactly the same way as an interface class. If you have put interface and interpreter into the same file, there is nothing more to do. If you have placed the interpreter in a separate file, let's say it is called `MyInterpreter.rb`, put the line

```
require "MyInterpreter"
```

into the file `SynInterfaces.rb` (in `Shalmaneser1.1/program/Pkg`), right after the interfaces (search for `require "TreetaggerInterface"`).

You do not need to choose an interpreter class explicitly in the experiment file. SHALMANESER chooses the first interpreter class it knows that covers all the systems (lemmatizer, POS-tagger, parser) you have chosen in the FRPREP experiment file.

We now list the methods offered by the `SynInterpreter` class. A star next to a method name means that it is recommended to overwrite the method. The default implementations are those provided by the `SynInterpreter` class. For an example of a fully specified `SynInterpreter`, have a look at the file `Tiger.rb`, class `Tiger`.

* `category` Generalizes over POS tags as well as constituent labels, returning strings like "verb", "noun", "prep". (see Table 8 for the complete list).

Default: Returns POS tag or constituent label as is.

* `relative_pronoun?` Returns `true` if the given syntactic node describes a relative pronoun.

Default: Always returns `false`.

`lemma_backoff` Returns the lemma for the given node (`nil` for nonterminals).

Abbreviation	Description
adj	adjective (phrase)
adv	adverb (phrase)
card	numbers, quantity phrases
con	conjunction
det	determiner, including possessive/demonstrative pronouns etc.
for	foreign material
noun	noun (phrase), including personal pronouns, proper names, expletives
part	particles, truncated words (German compound parts)
prep	preposition (phrase)
pun	punctuation, brackets, etc.
sent	sentence
top	top node of a sentence
verb	verb (phrase)
nil	something went wrong

Table 8: Set of generalised phrase types

Default: Returns the "lemma" attribute of the node if present; otherwise, or if the lemma includes "unknown" (a lemmatizer error code for Treetagger), returns the word instead.

`pt` Phrase type (as string)

Default: Returns the POS for terminals, and the result of the `category()` method for nonterminals.

`simplified_pt` Returns the phrase type (as string), may simplify it.

Default: Returns the result of the `pt()` method.

* `particle_of_verb` Tries to find the particle of particle verbs.

Default: Always returns `nil`, i.e. "none found".

* `auxiliary?` Returns `true` if the given syntactic node describes an auxiliary.

Default: Always returns `false`.

* `modal?` Returns `true` if the given syntactic node describes a modal verb. *Default:* Always returns `false`.

`head_terminal` Given a syntactic node, this method returns the terminal node that represents the constituent head.

Default: The "head" XML attribute set by preprocessing contains a node's head word. The method chooses the first terminal in the node's yield that matches this "head" attribute.

* `voice` Determines the voice for verb nodes, returning either "active" or "passive"; for non-verbs, returns `nil`.

Default: Using the `category()` method to decide whether the node is a verb, the method returns "active" for all verbs and `nil` for all other nodes.

`gfs` Determines the dependents of a constituent, both the syntactic nodes and their dependent labels.

Default: Returns all children of the given nodes as dependents, using the edge labels in the graph as dependent labels. For PPs, the preposition is appended to the dependent label.

* `informative_content_node` For most constituents, this method returns the head node. For a PP, returns the embedded NP; for an SBAR or a VP, the embedded VP.

Default: Returns the first non-head child of the given node.

`verbs` Returns a list of the syntactic nodes of the current sentence that represent full verbs.

Default: Uses the method `category()` to determine verbs.

`governing_verbs` Returns a list of the syntactic nodes of the current sentence that are verbs and that dominate a given node.

Default: Uses the method `category()` to determine verbs.

`path_between` Determines the path between two given nodes.

Default: Chooses the shortest path between the two nodes in the syntactic parse graph.

`surrounding_nodes` Starting from a given node, this method returns a list of all reachable nodes with the paths leading to them.

Default: Traverses the graph starting from the given start node.

`relative position` Determines the relative position of a given node with respect to an anchor node: "left", "right" or "dom".

Default: Compares the words spanned by the terminal yields of the two nodes, trying to account for continuous as well as discontinuous constituents.

`leftmost_terminal` Returns the leftmost terminal in the yield of a given node.

Default: Returns the leftmost yield node.

`rightmost_terminal` Returns the rightmost terminal in the yield of a given node.

Default: Returns the rightmost yield node.

`preposition` If the given syntactic node represents a PP, the method returns the preposition as a string.

Default: Uses the `category()` method to detect PPs, then assumes that the lemma of the `head_terminal()` of a PP is the preposition.

`main_node_of_expression` Given a list of nodes probably representing a multiword expression, this method tries to select a single head node.

Default: Filters out auxiliaries, modals, and particles of particle verbs (using `auxiliary?`, `modal?`, and `particle_of_verb`). If still left with more than node, the method resorts to heuristics.

`max_constituents` Given a set of nodes, computes the maximal constituents that exactly cover them.

Default: Chooses the smallest set of graph nodes that exactly cover the terminal nodes in the yield of the given node set. When there are several smallest set, prefers nodes higher up in the syntactic parse graph.

* `prune?` Given a target node and another node that is a candidate for a semantic role of the target, returns `true` to signal that the candidate is not a good candidate and should be omitted from the search for roles.

Default: Always returns `false`.

6.3. Adding a new machine learning system

Adding a new machine learning system to FRED. Each learner interface in FRED is a class inheriting from `FredAbstractClassifier`. For an example of such a class, see `NaiveBayes.rb` in `Shalmaneser1.1/program/Pkg`. Figure 8 shows the general structure of a learner interface: The `classifier_name()` class method announces the name by which this learner can be chosen in the FRED experiment file. The method `start_writing_classifier()` prepares the object for training a classifier, and `handle_training_instances()` actually trains the classifier. `start_writing_classifier` accepts as a parameter the lemma that the classifier will be specific to (a string), plus optionally the sense that the classifier will be specific to, in case of binary classifiers (another string). Analogously, the method `start_using_classifier()` prepares the object for applying a trained classifier (with the same parameters as in `start_writing_classifier()`), and `handle_test_instances()` actually applies the classifier. `close_classifier()` allows the interface class to close all currently still open files from writing or reading the last classifier.

Both `handle_training_instances()` and `handle_test_instances()` receive as a parameter an object that has an `instances()` method returning a list of instance hashes, and an `each_instance()` method that yields instance hashes. Each instance hash is a hash with the following entries:

"sentid" → the sentence ID, a string
"lemma" → the lemma, a string
"sense" → the sense, a string
"features" → an array of `FredFeature` objects.

The `FredFeature` class offers access to a `feature_value`, `feature_type`, `feature_parameters` and `weight`. For a detailed description, see the file `FredConventions.rb` in `Shalmaneser1.1/program/Pkg`. The `handle_test_instances()`

is supposed to return a `ClassificationResult` object, defined in `ClassifierResult.rb`. `ClassificationResult` objects allow the user to record a sequence of results (one for each test item), each as an `InstanceResult` object. An `InstanceResult` records the result for a single test item: for each possible sense label, the system score (or confidence value) can be set using the method `add_sense_prob(sense, score)`. (The winning sense will be the one with the highest confidence value).

To integrate your new learner interface into SHALMANESER, you need to enter it into the file `ClassifierList.rb` in `Shalmaneser1.1/program/Pkg`. Here is how the file will look with your additions, supposing your learner interface class is called `MyFredClassifier` and resides in the file `MyFredClassifier.rb`:

```
require "NaiveBayes"
require "MyFredClassifier"

@@implemented_classifier_list = [
  [NaiveBayesClassifier.classifier_name(), NaiveBayesClassifier],
  [MyFredClassifier.classifier_name(), MyFredClassifier]
]
```

To choose your new learner in your FRED experiment file, use the line

```
classifier = my_strategy
```

assuming you have set the return value of `MyFredClassifier.classifier_name()` to be `"my_strategy"`. At the moment, FRED cannot read the directory in which the classifier resides from the experiment file; you will have to hard-code it into your learner interface. This will be changed in the next version of the system.

Adding a new machine learning system for ROSY. Each learner interface in ROSY is a class offering the methods shown in Figure 9. For examples of such a class, see `Mallet.rb` and `Timbl.rb` in `Shalmaneser1.1/program/Pkg`.

Upon initialization, the interface class receives the program path and program parameters entered in the ROSY experiment file. For training a classifier, the two methods `train()` and `write()` are relevant: `train()` is supposed to train a classifier on the data in the given input file, and `write()` is supposed to write the last trained classifier to the given classifier file. Likewise, the methods `read()` and `apply()` pertain to applying a classifier: `apply()` is supposed to classify the instances in the given input file with the classifier passed to `read()`.

The format of the input file passed to `train()` and `apply()` is:

- ASCII, one instance per line
- comma-separated features
- the last feature is the gold label.

The output file of `apply()` is supposed to contain one line per classified instance. The first word of each line (i.e. everything up to the first whitespace) is taken to be the label assigned by the classifier. An empty line signals that nothing has been assigned to the corresponding instance.

To integrate your new learner into SHALMANESER, you will need to edit the file `ML.rb` in `Shalmaneser1.1/program/Pkg`. Suppose your new learner is a class called `MyRosyClassifier` in a file called `SomeFile.rb`, and suppose you would like to access your new learner in the experiment file by the name `my-rosy-classifier`. Then you will have to edit the `@@learners` class variable of the class `Classifier` such that afterwards it looks like this:

```
@@learners = [
    ["timbl", "Timbl", "Timbl"],
    ["mallet", "Mallet", "Mallet"],
    ["malouf", "Malouf", "Malouf"],
    ["my-rosy-classifier", "SomeFile", "MyRosyClassifier"]
 ]
```

You can then choose your learner in your ROSY experiment file by the name you have used in the `@@learners` array. We have chosen the name `my-rosy-classifier`. Now suppose that the directory in which the learner itself resides is `cl_path`, and the learner takes another parameter `p=5`, then the experiment file entry would look like this:

```
classifier = my-rosy-classifier cl_path p=5
```

```
class MyParserInterface < SynInterfaceSTXML
  MyParserInterface.announce_me()

  def MyParserInterface.system()
    return "my-parser"
  end

  def MyParserInterface.service()
    return "parser"
  end

  def process_file(infilename, outfilename)
    # apply my-parser to infilename,
    # write result to outfilename
    %x{my-parser #{infilename} > #{outfilename}}
  end

  def to_stxml_file(infilename, outfilename)
    # infilename contains parser output.
    # Transform to SalsaTigerXML
    # and write the result to outfilename.

    ...
  end

  def MyParserInterface.standard_mapping(sent, tabsent)
    # If your parser changes tokenization, you will
    # have to overwrite the standard_mapping method,
    # which describes the mapping between parse tree terminals
    # and words of the parser input sentences.
    #
    # This is rather intricate; if you need to do something
    # about tokenization, drop us a mail.

    ...
  end
end
```

Figure 6: A generic interface class for a parser

```
class MyInterpreter < SynInterpreter
  MyInterpreter.announce_me()

  ###
  # names of the systems interpreted by this class:
  # returns a hash service(string) -> system name (string),
  # e.g.
  # { "parser" => "collins", "lemmatizer" => "treetagger" }
  def MyInterpreter.systems()
    return {
      "pos_tagger" => "my-tagger",
      "parser" => "my-parser"
    }
  end

  ###
  # names of additional systems that may be interpreted by this class
  # returns a hash service(string) -> system name(string)
  # same as names()
  def MyInterpreter.optional_systems()
    return {
      "lemmatizer" => "my-lemmatizer"
    }
  end

  #...
end
```

Figure 7: A minimal generic interpreter class

```
class MyFredClassifier < FredAbstractClassifier

  def MyFredClassifier.classifier_name()
    "my_strategy"
  end

  def start_writing_classifier(lemma, sense="")
    # prepare for training a classifier
    # for the given lemma (and possibly sense)
  end

  def start_using_classifier(lemma, sense="")
    # prepare for using the classifier
    #  for the given lemma (and possibly sense)
  end

  def close_classifier()
    # close files
  end

  def handle_training_instances(reader)
    # train a classifier for the pre-set lemma (and sense)
    # on the given data.
  end

  def handle_test_instances(reader)
    # apply the classifier for the pre-set lemma (and sense)
    # to the given data. return results as a ClassifierResult object.
  end
```

Figure 8: A generic learner interface class for FRED

```
class MyRosyClassifier

  def initialize(program_path,parameters)
  end

  def train(infilename)
  end

  def write(classifier_file)
  end

  def exists?(classifier_file)
  end

  def read(classifier_file)
    # return true iff reading the classifier has had success
  end

  def apply(infilename,outfilename)
    # return true/false
  end
```

Figure 9: A generic machine learner interface class for ROSY

7. Bugs

Bug reports. If you discover a bug in the system, please notify us:

Katrin Erk: `erk@coli.uni-sb.de`

Sebastian Pado: `pado@coli.uni-sb.de`

It is crucial for us to get a description of the behaviour you consider a bug that is as detailed as possible. Since SHALMANESER uses quite a number of external components, we have seen a number of errors which are generated not by SHALMANESER, but by one of the modules. Two examples are:

Segmentation faults. Segmentation faults during the execution of SHALMANESER originate from the Ruby language itself, not from SHALMANESER. They occur sporadically, and usually never happen twice at the exact same spot. The only thing we can suggest here is: If possible, use Ruby 1.8.3 or 1.8.4 rather than 1.8.2; the newer versions seem to run much more smoothly.

Exceptions "Array out of bounds". `mallet`, the Java-based machine learning system, cannot deal with class labels in the test data unseen in the training data, and will respond with an exception. This should not happen if you have patched `mallet` successfully during setup. If you get the error anyway, it appears that your `mallet` system is unpatched. (try `grep -r pado *` – if you don't find anything, that is indeed the case). See Section 1.5 on how to patch `mallet`, or switch to a different learner.

Exceptions "java.io.FileNotFoundException (Permission denied)". textttmallet, the Java-based machine learning system, needs the training vectors to properly encode testing vectors; unfortunately, it insists on writing the training vectors back to the file afterwards. If the respective files are not writeable, you get this error. We might add a second `mallet` patch in the future, but currently the only workaround is to make sure that the `classifier` directory is writeable for all users, e.g. by calling `chmod -R g+rwX classifiers`.

8. Change Log

[v1.0] First SHALMANESER release

[v1.1] Second SHALMANESER release: bug fixes, reading FNCorpusXML, deeper frame representations, and FrameNet 1.3 classifiers

References

Collins, M. (1997). Three generative, lexicalised models for statistical parsing. In *Proceedings of ACL/EACL 1997*, Madrid, Spain, pp. 16–23.

Daelemans, W., J. Zavrel, K. van der Sloot, and A. van den Bosch (2003). Timbl: Tilburg memory based learner, version 5.0, reference guide. Technical Report ILK 03-10, Tilburg University. Available from `http://ilk.uvt.nl/downloads/pub/papers/ilk0310.ps.gz`.

Dubey, A. (2005). What to do when lexicalization fails: parsing german with suffix analysis and smoothing. In *Proceedings of ACL 2005*, Ann Arbor, Michigan.

Erk, K. (2005). Frame assignment as word sense disambiguation. In *Proceedings of IWCS 2005*, Tilburg, The Netherlands.

Erk, K. and S. Pado (2004). A powerful and versatile XML format for representing role-semantic annotation. In *Proceedings of LREC 2004*, Lisbon, Portugal.

Erk, K. and S. Pado (2006). Shalmaneser – a flexible toolbox for semantic role assignment. In *Proceedings of LREC-2006*, Genoa, Italy.

Erk, K. and S. Padó (2005). Analysing models for semantic role assignment using confusability. In *Proceedings of EMNLP 2005*, Vancouver, BC, pp. 668–675.

Gildea, D. and D. Jurafsky (2002). Automatic labeling of semantic roles. *Computational Linguistics 28*(3), 245–288.

Lin, D. (1993). Principle-based parsing without overgeneration. In *Proceedings of ACL-93*, Columbus, OH, USA.

McCallum, A. K. (2002). Mallet: A machine learning for language toolkit. http://mallet.cs.umass.edu.

Schmid, H. (1994). Probabilistic part-of-speech tagging using decision trees. In *Proceedings of NeMLaP 1994*.

Xue, N. and M. Palmer (2004). Calibrating features for semantic role labeling. In *Proceedings of EMNLP 2004*, Barcelona, Spain.

A. Sample experiment files

```
# ID identifying this experiment and all its data
# please do not use spaces inside the experiment ID
prep_experiment_ID = my_experiment

# location of input data, for storing finished
# preprocessed corpora
directory_input = /home/marie/all_salsa_data/
directory_preprocessed = /home/marie/all_salsa_data_preprocessed/

# experiment data description
language = de
origin   = SalsaTiger
format   = SalsaTigerXML
encoding = iso

# preprocessing
do_lemmatize = true
do_postag = true
do_parse = true

# directory where frprep puts its internal data
frprep_directory = /home/marie/tmp/frprep/

# Location of tools and resources used by Fred:
# part of speech tagging and lemmatization

lemmatizer = treetagger
lemmatizer_path = /proj/software/treetagger/cmd

# parser: location,
# maximum no. of sentences in a parse file,
# maximum sentence length to be parsed
parser = sleepy
parser_path = /proj/software/sleepy/
parser_max_sent_num = 2000
parser_max_sent_len = 80
```

Figure 10: Experiment file `exp_preproc.salsa`: Preprocessing of German data

```
# ID identifying this experiment and all its data
experiment_ID = my_experiment

# directories: where fred output is written
# and where classifiers are trained / searched for testing
directory_output = /home/marie/fred_output/
classifier_dir = /home/marie/classifiers/de/fred/my_exp/

# frprep experiment files
preproc_descr_file_train = exp_preproc.salsa
preproc_descr_file_test = exp_preproc.mytext

# Enduser mode?
enduser_mode = false


##################
# Features

feature = context 1.0 0.85
feature = ngram 1.6
feature = grfunc 1.1
feature = syn 4.0
feature = synsem 2.2

# feature dimensions:
feature_dim = word
feature_dim = lemma
feature_dim = pos

##################
# Featurization parameters

# use words from adjacent sentences?
# window size = number of adjacent sentences to consider
# set to 0, or don't set at all, in order not to use adjacent sentences
window_size = 0
```

Figure 11: Frame assignment: Part 1 of experiment file `exp_fred.salsa`

```
#################
# classifier parameters

# classifier type
classifier = naive_bayes

# classify all targets you know in test data?
# if false, use only the SalsaTigerXML targets
apply_to_all_known_targets = true

# Naive Bayes classifier:
# smoothing by giving some probability mass to unseen events:
# use lambda as parameter for Lidstone's law
smoothing_lambda = 0.55

#################
# tools and resources for FRED

# location of complete Tiger corpus in XML format
# needed only if origin = SalsaTiger and window_size > 0
tigerfile = %PATH%

# directory where Fred puts its internal data
fred_directory = /tmp/marie

#################
# Fred internal settings

<omitted>
```

Figure 12: Frame assignment: Part 2 of experiment file `exp_fred.salsa`

```
# Experiment ID:
experiment_ID = my_experiment

# Enduser mode?
enduser_mode = false

# frprep preprocessing experiment files
preproc_descr_file_train = exp_preproc.salsa
preproc_descr_file_test = exp_preproc.mytext

# input and output directory:
directory_input_train = /home/marie/all_salsa_data_preprocessed/
directory_input_test = /home/marie/fred_output/
directory_output = /home/marie/autoframe_done/
classifier_dir = /home/marie/classifiers/de/rosy/my_exp/

##
# Pruning: Identify constituents that are very unlikely as roles
prune = prune

# data adaptation: idealized FEs?
fe_syn_repair = true

# include relative clauses that have been left off the FE?
fe_rel_repair = false

# argrec classification by which category?
xwise_argrec = frame
xwise_arglab = frame
xwise_onestep = frame

# assume_argrec_perfect: can be set to true
#   to perform the arglab (argument labeling) step
#   separately on ''gold standard boundaries''
assume_argrec_perfect = false

# classifier
classifier = timbl /proj/software/Timbl5/Timbl
```

Figure 13: Role assignment: Part 1 of experiment file `exp_rosy.salsa`

```
# features
feature = pt_path
feature = gf_path
feature = path
feature = path_length
feature = pt_combined_path
feature = gf_combined_path
feature = combined_path
feature = pt_partial_path
feature = gf_partial_path
feature = partial_path
feature = pt_gvpath
feature = gf_gvpath
feature = gvpath
feature = ancestor_rule
feature = relpos
feature = pt
feature = gf
feature = gf_fn
feature = father_pt
feature = frame
feature = target
feature = target_pos
feature = target_voice
feature = gov_verb
feature = prep
feature = const_head
feature = const_head_pos
feature = icont_word
feature = firstword
feature = lastword
feature = leftsib
feature = rightsib
feature = worddistance
feature = ismaxproj
feature = nearest_node
feature = prune

# rosy data directory
data_dir = /home/marie/tmp/rosy/

########################
# rosy internal data - please don't change

<omitted>
```


Figure 14: Role assignment: Part 2 of experiment file `exp_rosy.salsa`

B. Accessing SalsaTigerXML-encoded sentences as graphs: The SalsaTigerRegXML package

This section describes the `SalsaTigerRegXML.rb` package (located in `Shalmaneser1.1/program/Pkg`), which offers graph-based access to SalsaTigerXML-encoded sentences. The package offers classes that describe a syntactically and semantically analyzed sentence as a collection of **trees with edge labels**. The package takes a node-centered view of trees: The objects offered by the package each describe *one tree node*, which, via methods to access the parent and children of the node, offers access to all the other nodes of the tree.

The syntactic structure of the sentence is described as one tree. Each frame (i.e., each predicate with its semantic roles) is described as another tree on top of the syntactic structure: The frame is described as one tree node, the children of which are semantic roles. The children of a semantic role node are nodes of the syntactic structure. The individual frame trees are independent of each other.

As an example, take the sentence in Figure 1.

- The **syntactic structure** of the sentence is represented as a tree. Each node of the tree is a **SynNode** object. Edges in the tree correspond to edge labels given by the parser in question (for a Collins parse, there are no edge labels).
- Each **frame** of the sentence is represented by one **FrameNode** object. For the SELF_MOTION frame, the FrameNode object has two children, which are **FeNode** objects. The edges to these two children are labeled *target* and SELF_MOVER.

  The SELF_MOVER FeNode represents the SELF_MOVER role. It has a single child: a SynNode representing the terminal *I* of the parse tree.

  The *target* FeNode represents the edge leading to the predicate. It, too, has a single child, the SynNode representing the terminal *Creeping*.

  Likewise, there is one FrameNode object each for the frames PATH_SHAPE and PERCEPTION_ACTIVE.
- The **sentence** as a whole is represented as a **SalsaTigerSentence** object. It has methods for accessing the syntactic structure as well as the frames.

Note that in principle, an FeNode can have more than one child, i.e. a role can point to several nodes of the syntactic structure (which indicates, mostly, that syntactic and semantic analysis do not coincide). Also, it is possible that a FrameNode has several FeNode children of the same name. This is another way of making a role point to several syntactic nodes. This second mode of describing "split" roles is necessary to model vagueness and ambiguity.

We now describe the most important classes offered by the `SalsaTigerRegXML` package.

The XMLNode class. XMLNode is a basic class from which all other classes of the package inherit. It describes a tree node that has some XML representation. Table 9 shows its most important methods.

Each node has an ID, which is supposed to identify the node uniquely (the object itself has no possibility to check this, though): node comparison relies solely on this node ID. An XMLNode is a node in a tree with edge labels and hence offers access to ancestors and descendants. Also, an XMLNode is a node that has an XML representation, which can be accessed using `get()`. Adding a child to the node adds an appropriate element to the XML representation, and XML attributes can be set, accessed and deleted directly via `set_attribute, get_attribute, del_attribute`.

The SalsaTigerXMLNode class. The SalsaTigerXMLNode inherits from XMLNode. While the XMLNode is independent of any specific XML representation, the SalsaTigerXMLNode represents a tree node that has a SalsaTigerXML representation. It may be a node of the syntactic or of the semantic structure. Table 10 shows the most important additional methods that this class offers: A SalsaTigerXMLNode can answer question on what kind of node in the syntactic or the semantic part of the sentence analysis it constitutes. Furthermore, it has a `to_s()` method that returns the sentence words dominated by the node, and it has a `terminals_sorted()` method, which returns the terminals dominated by the node, sorted in order of appearance of the words in the sentence.

The SynNode class. The SynNode class describes a node of the syntactic structure. It inherits from SalsaTigerXMLNode, adding the methods shown in Table 11.

The FrameNode class. The FrameNode class describes a frame node: the root of a tree of the semantic analysis. Its children are FeNode objects, i.e. the inherited `children()` and `each_child()` methods return FeNode objects. Each of them describes one semantic role, except for one distinguished child called the *target*, which points to the predicate that evokes the frame. A FrameNode also has a *frame*, which indicates a sense for the predicate. FrameNode inherits from SalsaTigerXMLNode, adding the methods shown in Table 12.

The FeNode class. The FeNode class describes one semantic role. Its parent is a FrameNode object, and its children are SynNode objects, i.e. the inherited `children()` and `each_child()` methods return FeNode objects, the syntactic nodes to which the role points. FeNode inherits from SalsaTigerXMLNode, adding the methods shown in Table 13.

The SalsaTigerSentence class. The SalsaTigerSentence class represents a complete sentence with syntactic and semantic analysis. A SalsaTigerSentence object is initialized with the XML representation for the sentence as a string. It provides access to nodes of the syntactic and the semantic analysis, and it allows its user to add and delete nodes in either part of the sentence representations. In addition to a syntactic and semantic analysis, a sentence may possess *sentence flags*, for example to record difficulties in annotation. The SalsaTigerSentence class also provides two `max_constituents` methods. Given a list of SynNode objects, they search for the maximal syntactic constituents that exactly cover the given nodes. While the first `max_constituents` method does exactly this, the second method allows for constituents that may cover more or less than the given node list, provided they fulfil certain conditions.

SalsaTigerSentence	
Initialization	
`new(xml)`	Initializes the SalsaTigerSentence object. Parameter: XML representation as string.
`empty_sentence(id)`	Class method, returns a SalsaTigerSentence object representing an empty sentence with the given sentence ID.
String representation	
`get()`	Returns the current XML reprentation of the sentence as a string.
`get_syn()`	Returns the current XML reprentation of the *syntactic part* of the sentence as a string.
`to_s()`	Returns the words of the sentence as a string (concatenated by spaces), with the words in the right order.
Access to nodes	
`terminals()`, `terminals_sorted()`, `each_terminal()`, `each_terminal_sorted()`	These methods provide access to the terminal SynNodes of the sentence, either by returning them as a list (the first two methods) or via an iterator. The two ... `sorted()` methods sort terminals by the order of appearance in the sentence, assuming that terminal node IDs end in numbers and that terminals occurring later in the sentence have higher numbers.
`nonterminals()`, `each_nonterminal()`	These methods provide access to the nonterminal SynNodes of the sentence, by returning them as a list (first method) and via an iterator (second method).
`syn_nodes()`, `each_syn_node()`	These methods provide access to all nonterminal SynNodes of the sentence, by returning them as a list (first method) and via an iterator (second method).
`frames()`, `each_frame()`	These methods provide access to all FrameNodes of the sentence, by returning them as a list (first method) and via an iterator (second method).
`syn_roots`	This method returns a list of all *root* SynNodes of the sentence (allowing for the possibility that the syntactic representation of a sentence may not be a coherent tree but a collection of trees).
`syn_node_with_id(id)`, `sem_node_with_id(id)`	These two methods look for a node given its node ID, either a syntactic node (first method) or a frame or role node (second method). If the node is found, it is returned, otherwise the method returns `nil`. Parameter: the node ID as a string.

Adding or removing nodes	
`add_syn(t, c, w, p, id?)`	Adds a syntactic node and returns it, although without linking it to the other syntactic nodes. This can be done subsequently using the `add_child()` and `add_parent()` methods of SynNode. Parameters: `t`=type, either "t" for terminal or "nt" for nonterminal; `c`=constituent label (string, `nil` for terminals); `w`=word (string, `nil` for nonterminals); `p`=part of speech (string, `nil` for nonterminals); `id`= node ID (string, optional parameter; if omitted, an ID will be generated from sentence ID and system time)
`add_frame(l, id?)`	Adds a frame node and returns it. To add roles and link them to the syntactic structure, use `add_fe` below. Parameters: `l`=label, the frame name (string); `id`=node ID (string, optional parameter; if omitted, an ID will be generated from sentence ID and system time)
`add_fe(f, l, ch, id?)`	Adds a semantic role and returns it. The semantic role is linked to its frame and the syntactic nodes that it points to directly through this method. Parameters: `f`=FrameNode that the role belongs to; `l`=label, the role name (string); `ch`=children, a list of SynNode objects; `id`=node ID (string, optional parameter; if omitted, an ID will be generated from sentence ID and system time)
`remove_syn(n)`	Removes a node of the syntactic structure. Its children will be linked to the node's former parent. Parameter: SynNode object.
`remove_frame(n)`	Removes a FrameNode. Parameter: FrameNode object.
`remove_fe(n)`	Removes a semantic role. Parameter: FeNode object.
`remove_semantics()`	Removes the complete semantic analysis of a sentence.
Sentence flags	
`flags()`	Returns the list of sentence flags as a list of hashes. Each flag-hash has keys "type", "param" and "text" that together describe the flag.
`add_flag(t, p, tx)`	Adds a sentence flag. Parameters: `t`=type, `p`=parameter, `tx`=text. All three parameters are strings. Parameters `p` and `tx` may be omitted.
Maximum constituents	
`max_constituents_for_nodes(l)`	Given a list `l` of nodes, the method determines the maximum constituents that exactly cover the nodes in the given list and returns them as a list of SynNode objects. Terminals representing punctuation are ignored.

`max_constituents_smc(l, b1, b2, p?)`

Given a list `l` of nodes, the method determines the maximum constituents that exactly cover the nodes in the given list and returns them as a list of SynNode objects. Terminals representing punctuation are ignored.

Parameters: `l`=nodelist: list of SynNode objects. `b1`:include single missing children? (Boolean) If *true*, the method will consider parent constituents when all of its children except one are already (potentially non-maximal) nodelist-covering nodes – provided the parent constituent has at least 3 children. `b2`: ignore empty terminals? (Boolean) If *true*, empty terminal nodes are ignored like punctuation. `p`: accept anyway? a Proc object, which may be omitted. The procedure `p` is called for nodes that are not nodelist-covering. If it decides to accept the node anyway, it will be counted as nodelist-covering. This procedure thus provides complete flexibility in extending `max_constituents_smc` by arbitrary additional acceptance criteria. The procedure is called with 3 arguments: a SynNode `n` that would not normally be a nodelist-covering node; a list of `n`'s children that are nodelist-covering; and a list of `n`'s children that are node nodelist-covering. If the procedure returns *true*, `n` will be regarded as nodelist-covering and used in the rest of the computation of maximal constituents from then on.

XMLNode	
Node properties	
`id()`	Node ID
`==(n)`	Node comparison relies *solely* on the node ID
Ancestors	
`parent(),parent_label()`	Return the parent node / the edgelabel leading to the parent
`ancestors()`	All ancestors of the node (excluding the node itself)
`add_parent(n, e)`, `set_parent(n, e)`	Add/change the parent node. Parameters: both the parent node and the edgelabel leading to the parent
Descendants	
`children(),each_child()`	Children nodes, either as a list or via an iterator.
`child_label(n)`	Edgelabel leading to the given child node. Parameter: child node
`children_by_edge-labels(el)`	Given a list `el` of edge labels, returns the list of children that can be reached via these edgelabels
`add_child(n, e)`, `remove_child(c, e)`	Add or remove child node. Parameters: child node, edge label
`outdeg()`	Out-degree: number of children
`yield_nodes()`	Returns a list of all the *terminal* nodes that the current node dominates.
`descendants()` `descendants_by_edge-labels(el)`	All descendants of the node (excluding the node itself). `descendants_by_edgelabels`: in analogy to `children_by_edgelabels`, uses only edges whose labels are included in the given set.
XML	
`set_attribute(n, v)`, `get_attribute(n)`, `del_attribute(n)`	Set, get, or delete an attribute of the XML representation of the node. Parameters: name of the attribute(string); for `set_attribute` also: value of the attribute
`get()`	Returns the XML representation of the node as a string.

Table 9: The XMLNode class: most important methods

SalsaTigerXMLNode	
Node type tests	
`is_terminal?`	True if this is a syntactic node representing a terminal.
`is_nonterminal?`	True if this is a syntactic node representing a nonterminal.
`is_splitword?`	SalsaTigerXML allows terminal nodes to be split, e.g. for a detailed analysis of compounds. The pieces of such a split are splitword nodes. This method returns true for this kind of nodes.
`is_syntactic?`	True for terminal, nonterminal, and splitword nodes.
`is_frame?`	True for nodes representing frames (describing predicate sense and semantic roles).
`is_target?`	True for the "target" child of a frame node, which points to the predicate of the frame.
`is_fe?`	True for all semantic role nodes, the children of a frame node.
`is_outside_sentence?`	SalsaTigerXML allows for semantic annotation to cross the sentence boundary. If a semantic role points to a syntactic node outside the current sentence, `is_outside_sentence?` will return true for that syntactic node.
Other methods	
`to_s()`	Returns a string, the words belonging to this node separated by whitespace. The words belonging to this node are the words attached to the terminal nodes dominated by this node. Word order is determined using `terminals_sorted()`.
`terminals_sorted()`	Returns the terminal nodes (SynNode objects) dominated by this node, ordered by the order of the words attached to them. Word order is determined using node IDs, working on the assumption that node IDs of terminals end in numbers and that higher numbers indicate words that occur later in the sentence.
`sid()`	Returns the ID of the sentence that this node belongs to, working on the assumption that the node ID up to the first underscore is identical to the sentence ID.

Table 10: The SalsaTigerXMLNode class: most important additional methods

SynNode	
`part_of_speech()`	Returns the part of speech (as a string) for a terminal node.
`category()`	Returns the constituent label (as a string) for a nonterminal node.
`word()`	Returns the word (as a string) for a terminal node.
`is_punct?`	Returns *true* for terminals representing punctuation.

Table 11: The SynNode class: most important additional methods

FrameNode	
`name()`	Returns the frame name as a string.
`target()`	Returns the child that stands for the target (as an FeNode)
`add_flag(s)`	Frame nodes can bear flags, e.g. to indicate metaphor. This method adds a flag. Parameter: the flag, a string.
`remove_flag(s)`	This method removes a flag of the given name, if it exists. Parameter: the flag, a string.

Table 12: The FrameNode class: most important additional methods

FeNode	
`name()`	This method returns the name of the semantic role (e.g. SELF_MOVER or *target*) as a string.
`add_flag(s)`	Like FrameNode objects, FE nodes can bear flags. This method adds a flag. Parameter: the flag, a string.
`remove_flag(s)`	This method removes a flag of the given name, if it exists. Parameter: the flag, a string.

Table 13: The FeNode class: most important additional methods