

Robust Natural Language Text Processing with of SMES

A brief description and user manual

Günter Neumann
DFKI
Saarbrücken, Germany

November 26, 2003

1 Introduction

This is a brief description of using SMES, a a robust and fast NL text processor and information extraction core system for German. A detailed description of SMES can be found in [Neumann *et al.*, 1997], [Neumann *et al.*, 2000], and [Neumann and Piskorski, 2002]. Important properties of SMES are:

- a clean separation between linguistic and template processing (as described in [Neumann and Mazzini, 1998]); particularly this means that SMES can be used in several ways:
 - for performing shallow NL text processing,
 - for performing **nearly deep large scale NL text analysis** only
 - for building naive IE-applications
 - combining both: building advanced IE systems
- replacement of the bidirectional FST through a *cascaded divide-and-conquer parser*

- powerful morphological interface which supports efficient feature relaxation and unification.

SMES has a high degree of modularity. Every major component — the tokenizer, the morphology, the chunk parser, the clause parser — can be used in isolation. In order to make this modularity as transparent as possible, there exists common function interfaces to each individual module.

We will now start describing the system. We will focus on the core functionality, i.e., a description will be given of how SMES can be used as *shallow text processor*. For the rest of the paper, we assume that a SMES user has basic knowledge of Unix, Emacs and Common Lisp.

2 Platform

Note that all information concerning installation are based on the hard- and software configuration as set up at the LT-Lab of DFKI.

SMES should be platform independent as far as Common Lisp is supported. Note that to run SMES on larger documents you really need to compile SMES first, before you load it.

Compilation and loading is very simple. In the subdirectory SYSTEMS you will find the file SMES-SYS.lisp. This is the only file you need to change. Have a look into it and change the two variables `*smes-sys-root*`, and `*image-pathname*`. This should be enough. Then call function `(compile-smes)` to compile the system, `(load-smes)` to load the binary stuff and `(dump-smes)` to create an image file. This should really everything you need to know (modulo data-source variables, grammar writing and so).

3 Initializing and exiting SMES

Initialization Enter `(INIT-SMES)` if your are in the USER package — or `(USER::INIT-SMES)` if not — in order to initialize SMES. This will automatically jump directly into the package SMES, the main package of SMES.

Exiting Enter `(EXIT-SMES)` from package SMES in order to exit SMES. Note that this will not exit the current Lisp image.

4 General notes on input and output

As already mentioned, SMES has a high degree of modularity. In order to make it transparent to the user, there exists a common function interface mechanism to each major module.

General form of input All functions accept as (minimal) input a string which represents either

- the ascii-text to be analyzed, or
- a pathname to a file containing the ascii-text.

In order to distinguish between the two input possibilities, there exists two sorts of functions, viz. XXX-FROM-STRING which calls the module XXX for a string representing an ascii-text and XXX-FROM-FILE where the input string is interpreted as a pathname.

Global parameter for setting the input directory The global variable `SMES::*CORPORA-DIR*` bounds the pathname of a directory which contains text documents to be processed. The value must be a pathname. Simply entering the variable name will return the current used pathname. In order to change the current value — say to `/home/cl-home/krieger/tmp/` — enter

```
(setq smes::*corpora-dir*  
      (pathname "/home/cl-home/krieger/tmp/"))
```

Now, assuming you want to perform a morphological analysis of the content of a file named `/home/cl-home/krieger/tmp/morph.txt` you may enter

```
(morph-from-file "morph.txt")
```

or

```
(morph-from-file "/home/cl-home/krieger/tmp/morph.txt")
```

Note that the default file type in SMES is `.TXT`, so that you might even call

```
(morph-from-file "morph")
```

General form of output All -FROM- functions return a list of all found results. Each result is also a list, but the concrete form may differ from module to module.

HTML-based output Additionally, the results of the chunk parser can be mapped to HTML-marked up expressions which are stored in a temporary file. All functions which are able to map their resulting output to an HTML-format are named XXX-HTML-STRING and XXX-HTML-FILE, respectively. Note that constructing the HTML-file is performed via a side-effect and that these functions return NIL has their result. Thus they are basically used for creating marked-up text files.

With help of the variable `*suppress-features*` you can define, which features should not be print in the HMTL output. If all information should be used, then set it to NIL. For an example setting, see file `../pd-smes/new-parse/smes2html.lis`

5 The main modules

5.1 Tokenization

The tokenizer is called via the following two functions:

1. (SCAN-FROM-STRING <A TEXT STRING>)
2. (SCAN-FROM-FILE <A PATHNAME STRING>)

The tokenizer returns a list of lists where each list represents a sentence. A sentence is simply a list of recognized tokens, where the last token belongs to an interpunction sign (one of . ? or !). Note that the tokenizer only performs a very simple recognition of sentence boundaries (actually completely context-free), because proper sentence boundary recognition will take place during chunk parsing. Example:

```
(scan-from-string "Ich gehe ! Peter sagt: ich auch. ") yields
```

```
((("Ich" "gehe" "!")  
 ("Peter" "sagt" (:SPECIAL . ":") "ich" "auch" "."))
```

Word forms are represented as strings by the tokenizer; every other token, e.g., special characters or certain fragments, like time or date expression, is represented as a list). An EBNF description of the output structure can be found in appendix A.

5.2 Morphology

Basic functions Morphological processing is performed by an extended version of Morphix [Finkler and Neumann, 1988] called Morphix++. Morphix++ is called via two functions:

1. (MORPH-FROM-STRING <A TEXT STRING> :TIME <BOOLEAN>)
2. (MORPH-FROM-FILE <A PATHNAME STRING> :TIME <BOOLEAN>)

where :TIME is a boolean parameter (the default value is NIL). When its value is T, it outputs the real-time (i.e., including garbage collection and operation system calls) for each module, separately.

The morphology returns a list of lists where each lists corresponds to a sentence. Each morphological analysed word form (word forms are represented as strings by the tokenizer; every other token, e.g., time or date expression is represented as a list) are represented as a triple of the form $\langle stem, inflection, pos \rangle$, where *stem* is a string or a list of strings (in the case of compounds), *inflection* is the inflectional information, and *pos* is the part of speech. Example (abbreviated where convenient):

(morph-from-string "Dem Ingenieur ist nichts zu schwoer. ") yields

```
((("Dem"
  ("d-det"
    (((:TENSE . :NO) ... (:GENDER . :M) (:NUMBER . :S)
      (:CASE . :DAT))
    (((:TENSE . :NO) ... (:GENDER . :NT)
      (:NUMBER . :S) (:CASE . :DAT)))
    . :DEF))
  ("Ingenieur"
    ("ingenieur"
```

```

      (((:TENSE . :NO) ... (:CASE . :NOM))
       (((:TENSE . :NO) ... (:CASE . :DAT))
        (((:TENSE . :NO) ... (:CASE . :AKK)))
       . :N))
("ist"
 ("sei"
  (((:TENSE . :PRES) ...
   (:NUMBER . :S) (:CASE . :NO)))
  . :AUX))
("nichts" ("nichts" NIL . :PART))
("zu" ("zu" NIL . :SUBORD)
 ("zu"
  (((:TENSE . :NO) ... (:CASE . :DAT)))
  . :PREP))
("schwoer"
 ("schwoer"
  (((:TENSE . :NO) (:FORM . :IMP) ...
   (:NUMBER . :S) (:CASE . :NO)))
  . :V))
("." ("." NIL . :INTP)))

```

Form of inflection Morphix++ has a very flexible output interface allowing for different representations of the inflectional information of an analysed word. In SMES we use a feature vector representation in disjunctive normal form (DNF). Although Morphix++ has a rich tag set (see appendix B) we only make use of a small subset of inflectional features, namely:

```
:TENSE :FORM :PERSON :GENDER :NUMBER :CASE
```

Note that in case a wordform misses one of the features (e.g., a noun form has no :TENSE feature) the missing feature is returned with the special value :NO. During chunk parsing and agreement checking (see next section) this value is handled as an anonymous variable.

Some useful settings The following variables can be used to parametrize morphological processing within SMES. Note that these variables are global so that their re-setting will affect every function which uses morphology as a subcomponent, e.g., the chunk parser:

variable name SMES::**APPLY-WRITING-RULES**, **range** T or NIL, **default** T, **documentation** triggers application of case-sensitive rules; if set to T, then words are disambiguated on the basis of upper/lower writing (e.g., only nouns are written with an initial upper letter in German);

variable name MO::**HANDLE-UNKNOWN**, **range** T or NIL, **default** T, **documentation** triggers robust processing of compounds; if set to T then compounds are recognized by means of longest matching substring found in the lexicon; e.g., the word “adfadfeimer” will return result for “eimer” assuming that “adfadf” is no legal lexical stem;

variable name MO::**ALL-COMPOSITA**, **range** T or NIL, **default** NIL, **documentation** if NIL perform decomposition according to longest matching lexical entries; otherwise, all possible decompositions are returned.

6 Part-of-speech disambiguation

Morphological ambiguous readings are disambiguated wrt. part-of-speech using *case-sensitive rules* which are applied after morphological processing but before chunk parsing.

Generally, only nouns (and proper names) are written in standard German with an capitalized initial letter (e.g., “der Wagen” *the car* vs. “wir wagen” *we venture*). Since typing errors are relatively rare in press releases (or similar documents) the application of case-sensitive rules are a reliable and straightforward tagging means for the German language. Of course, the case-sensitive rules should only be applied, if there is enough evidence that the document’s author actually followed typical German spelling rules. For that reason, the case-sensitive rules are only applied, if the value of a statically driven lookahead function $\text{TRIGGER-WR}(k)$ exceeds some threshold y applied on the k -first tokens computed by the `TEXT TOKENIZER`. Actually, the method counts all strings s (i.e., possible wordforms) that do not follow the dot sign and all lower-case strings ls . Then if $s > 0$ and $ls/s \leq y$ then $\text{TRIGGER-WR}(k)$ returns T (meaning that the case-sensitive rules will be applied); otherwise NIL is returned which means that no disambiguation on the basis of cases-sensitivity will be made.

Some useful variables The following variables are used to set the parameters for the case-sensitive rules:

variable name *APPLY-WRITING-RULES* **range** BOOLEAN **default** T **documentation** if T then case-sensitive rule are active; inactive if set to NIL;

variable name *CHECK-OE-STYLE* **range** BOOLEAN **default** T **documentation** if T apply function TRIGGER-WR(k); if its value is T and *apply-writing-rules* is set to T then apply case-sensitive rules; if *check-oe-style* is set to NIL then apply in dependence of the current value of *apply-writing-rules*;

variable name *OE-UPPER-LIMIT* **range** INTEGER **default** 30 **documentation** length of the lookahead;

variable name *OE-THRESHOLD* **range** REAL **default** 0.85 **documentation** the threshold for activating the case-sensitive rules;

7 Chunk parsing

Chunk parsing is performed in three steps:

- recognition of phrases
- recognition of clauses
- recognition of grammatical functions

The first two steps are performed by finite state grammars. The last step is performed by means of a huge subcategorization lexicon (about 25.000 entries) and some general mechanisms. All three steps correspond to modules which can be called in isolation as described now. See [Neumann *et al.*, 2000] for details.

7.1 Recognition of phrases

Basic functions Phrase recognition is activated via the following functions (there also exists corresponding functions which create HTML-files of the

found phrases and are described separately below):

1. (FST-FROM-STRING <A TEXT STRING>) :FST <SUBGRAMMAR> :TIME <BOOLEAN>)
2. (FST-FROM-FILE <A PATHNAME STRING> :FST <SUBGRAMMAR> :TIME <BOOLEAN>)

where the optional parameter :TIME returns the realtime used by these functions. The parameter :FST is obligatory, where <SUBGRAMMAR> is the name of a phrasal subgrammar. The following table shows the list of the build-in subgrammars which are visible to the user (but note that SMES supports writing and integration of your own grammars):

Name	Description
NP-STAR	nominal phrases (NPs) only
MAIN	nominal and prepositional phrases (PPs) only; note that no PP-attachment is performed
TIME-DATE	complex time and date expressions
FIRMEN-TREIBER	company name information
NUMBER-TREIBER	currency expressions
ALL-FRAGS	all subgrammars (except firm recognition but plus verb group recognition) actually the result of this subgrammar is passed to the clause level

Some notes about the subgrammars' output structure Each subgrammar is expected to represent its resulting structures uniformly as feature value structures, together with its type and the corresponding start and end positions of the spanned input expression. We call these output structures *text items*. A subgrammar's individual output feature structure is expected to be the value of a feature SEM. Consider the following simple example, where we use the subgrammar named MAIN (see above):

```
(fst-from-string
  "Der Mann sieht die Frau mit dem Fernrohr." :fst 'main)
```

which yields (abbreviated where convenient):

```
(((:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
 (:AGR
  ((:TENSE . :NO) ... (:CASE . :NOM)))
 (:END . 2) (:START . 0) (:TYPE . :NP))
 (:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
 (:AGR
  ((:TENSE . :NO) ... (:GENDER . :F) (:NUMBER . :S)
   (:CASE . :NOM))
  ((:TENSE . :NO) ... (:GENDER . :F) (:NUMBER . :S)
   (:CASE . :AKK)))
 (:END . 5) (:START . 3) (:TYPE . :NP))
 (:SEM (:HEAD "mit")
  (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr")))
 (:SUB ((:SEM # #) (:AGR #)
  (:END . 8) (:START . 6) (:TYPE . :NP)))
 (:AGR
  ((:TENSE . :NO) ... (:GENDER . :NT) (:NUMBER . :S)
   (:CASE . :DAT)))
 (:END . 8) (:START . 5) (:TYPE . :PP)))
```

The resulting structures are interpreted as head/modifier structures. Note that the prepositional phrase (PP) “mit dem Fernrohr” has not been attached to one of the nominal phrases (NPs) since this cannot be deterministically decided without further analysis. Note further that the PP’s agreement information is fully disambiguated.

HTML-based functions The following two functions map the resulting output to some HTML format and store them into a file.

1. (FST-HTML-STRING <A TEXT STRING> :FST <SUBGRAMMAR> :TIME <BOOLEAN>)
2. (FST-HTML-FILE <A PATHNAME STRING> :FST <SUBGRAMMAR> :TIME <BOOLEAN>)

They behave in the same way as their “-FROM-” counterparts, with the notable difference that their return value is NIL. The resulting text items are stored in the file named

“~/tmp/smes-phrase-res.html”

This file will be automatically created if it does not exist (we are assuming the user has already created a directory called *tmp* under her home directory) or will overwrite the file if it already exists. This file — which will be superseded every time one of the “-HTML-” functions are called — can now be displayed with some web-browser, e.g., Netscape. The HTML-file consists of the marked-up text, where the markers are linked to a glossary. The glossary displays the internal text items in form of a feature matrix, where not all internal information is visualized in order to support readability.

7.2 Recognition of Clauses

Have a look into [Neumann *et al.*, 2000] in order to understand SMES divide-and-conquer strategy. It also explains the notion of topological field structure and underspecified dependency structure!

Basic functions Clause recognition is called on the result of phrase recognition (see previous subsection). You can call clause recognition via the following functions (where their “-HTML-” counterparts are described below):

1. (PARSE-FROM-STRING <A TEXT STRING> :TIME <BOOLEAN>)
2. (PARSE-FROM-FILE <A PATHNAME STRING> :TIME <BOOLEAN>)

where :TIME is a boolean parameter (the default value is NIL). When its value is T, it outputs the real-time (i.e., including garbage collection and operation system calls) for each module, separately.

These functions perform (in that order) tokenization, morphological analysis, phrasal recognition, and finally clause recognition. Note that in our current version, ALL-FRAGS is the subgrammar automatically called for performing phrase recognition. Furthermore note that clause recognition is processed with the same finite state tool, but with a specific clause grammar, which can be viewed as a set of sentence patterns.

HTML-based functions The following two functions map the resulting output to some HTML format and stores it into a file.

1. (PARSE-HTML-STRING <A TEXT STRING> :TIME <BOOLEAN>)
2. (PARSE-HTML-FILE <A PATHNAME STRING> :TIME <BOOLEAN>)

The analyzed text together with its parsed results are stored in the file named

“~/tmp/smes-parse-res.html”

Determining sentence boundaries Before clause recognition is called, the flat stream of recognized phrases is partitioned into a list of sentences according to a function MAKE-SENTENCE which is called after phrase recognition but before clause recognition. Then clause recognition is applied on each sentence individually. Each sentence corresponds to the list of all found phrases and is processed by means of the finite-state clause grammar simply named PARSER. Using this grammar the chunk parser tries to combine every phrase to build up a parse tree. If not all possible phrases fit into one structure, the parser returns a *partial* parse tree, such that a list consisting of the “longest” matching common sub-sentence and a list of all non-fitting remaining phrases is returned. We will simply call such a partial result an *incomplete parse tree*.

Some notes on the structure of parse trees Linguistically, clause recognition is performed by means of a dependency-based grammar, i.e., the resulting parse tree is actually a dependency tree. However, in the same way as we delay PP-attachment during phrase recognition, the dependency grammar only defines *upper bounds* for attachment, which is mainly defined by the head element. Since a dependency tree is potentially recursive, such trees are not totally flat, but coarse-grained wrt. attachment. We call such a dependency tree an *underspecified dependency tree*. Consider the following example:

```
(parse-from-string  
  "Der Mann sieht die Frau mit dem Fernrohr.")
```

which yields the following underspecified dependency tree (abbreviated where convenient):

```
(((:PPS
  ((:SEM (:HEAD "mit")
    (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr"))))
  (:SUB ...)
  (:AGR
    ((:TENSE . :NO) ... (:CASE . :DAT)))
  (:END . 8) (:START . 5) (:TYPE . :PP)))
(:NPS
  ((:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
  (:AGR
    ((:TENSE . :NO) ... (:CASE . :NOM)))
  (:END . 2) (:START . 0) (:TYPE . :NP))
  ((:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
  (:AGR
    ((:TENSE . :NO) ... (:CASE . :NOM))
    ((:TENSE . :NO) ... (:CASE . :AKK)))
  (:END . 5) (:START . 3) (:TYPE . :NP)))
(:VERB
  (:COMPACT-MORPH
    ((:TEMPUS . :PRAES) ... (:PERSON . 3)
    (:GENUS . :AKTIV)))
  (:MORPH-INFO
    ((:TENSE . :PRES) (:FORM . :FIN) ... (:CASE . :NO)))
  (:ART . :FIN) (:STEM . "seh")
  (:FORM . "sieht") (:C-END . 3) (:C-START . 2)
  (:TYPE . :VERBCOMPLEX))
  (:END . 8) (:START . 0) (:TYPE . :VERB-NODE)))
```

In this structure, the feature “:VERB” collects all information of the complex verb group which is the head of the sentence. “:PPS” collects all PPs and “:NPS” is a list of all dependent nominal phrases. An EBNF of the currently covered clausal expression can be found in Appendix D

7.3 Recognition of topological structure

SMES is very good in processing the topological structure of German sentences using only view linguistics background knowledge.

If you are interested only in the topological structure (with and without chunk parsing) you should call the above functions with additional keyword parameter `:out` and its two possible values `:t1` (without chunk parsing), `:t2` with chunk-parsing. Senseless example for *The little man who goes to school is still not allowed to drive a car.*

```
(parse-from-string "Der kleine Mann, der zur Schule geht, darf
noch nicht Auto fahren." :out :t1)
```

```
(((:SENT-MARKER . ".")
(:VF
("Der"
("d-det"
(((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :M)
(NUMBER . :S) (CASE . :NOM))
...
((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :NT)
(NUMBER . :P) (CASE . :GEN)))
. :DEF))
("kleine"
("klein"
(((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :M)
(NUMBER . :P) (CASE . :NOM))
...
((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :F)
(NUMBER . :S) (CASE . :AKK)))
. :A))
("Mann"
("mann"
(((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :M)
(NUMBER . :S) (CASE . :NOM))
...
((TENSE . :NO) (FORM . :NO) (PERSON . 3) (GENDER . :M)
(NUMBER . :S) (CASE . :AKK)))
. :N))
```

```

((:CONTENT
 (:MF
  ("zur"
   ("zur"
    (((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NO)
      (:NUMBER . :NO) (:CASE . :DAT)))
    . :PREP))
  ("Schule"
   ("schule"
    (((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :F)
      (:NUMBER . :S) (:CASE . :NOM))
    ...
    ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :F)
      (:NUMBER . :S) (:CASE . :AKK)))
    . :N)))
 (:VERB (:LENGTH . 1) (:END . 8) (:START . 7)
 (:MORPHO-INFO
 (:AGR
  ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 2) (:GENDER . :NO)
   (:NUMBER . :P) (:CASE . :NO))
  ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3) (:GENDER . :NO)
   (:NUMBER . :S) (:CASE . :NO)))
  (:FINIT . T) (:GENUS . :AK) (:TENSE . :PRE))
  (:STEM . "geh") (:FORM . "geht") (:TYPE . :VERB))
 (:LENGTH . 3) (:TYPE . :SPANNSATZ))
 (:LENGTH . 5) (:END . 8)
 (:REL-PRON
  ("der"
   ("d-det"
    (((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :M)
      (:NUMBER . :S) (:CASE . :NOM))
    ...
    ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
      (:NUMBER . :P) (:CASE . :GEN)))
    . :DEF)))
  (:START . 3) (:TYPE . :REL-CL))
  ("," ("," NIL . :INTP)))
 (:MF ("noch" ("noch" NIL . :PART)) ("nicht" ("nicht" NIL . :PART))
  ("Auto"

```

```

("auto"
  (((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
    (:NUMBER . :S) (:CASE . :NOM))
    ...
    ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
      (:NUMBER . :S) (:CASE . :AKK)))
  . :N)))
(:VERB (:LENGTH . 2) (:END2 . 14) (:START2 . 13) (:END . 10)
  (:START . 9)
  (:MORPHO-INFO
  (:AGR
    ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3) (:GENDER . :NO)
      (:NUMBER . :S) (:CASE . :NO))
    ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 1) (:GENDER . :NO)
      (:NUMBER . :S) (:CASE . :NO)))
    (:FINIT . T) (:GENUS . :AK) (:TENSE . :PRES))
  (:SCOPE (:LENGTH . 1) (:END . 14) (:START . 13)
  (:MORPHO-INFO
  (:AGR
    ((:TENSE . :SUBJUNCT-1) (:FORM . :FIN) (:PERSON . :ANREDE)
      (:GENDER . :NO) (:NUMBER . :P) (:CASE . :NO))
    ...
    (:STEM . "fahr") (:FORM . "fahren") (:TYPE . :VERB))
    (:STEM . "duerf") (:FORM . "darf fahren") (:TYPE . :MODVERB))
  (:END . 15) (:START . 0) (:LENGTH . 14) (:TYPE . :KERNSATZ)))

```

Now, the same example with chunk parsing:

```

(parse-from-string "Der kleine Mann, der zur Schule geht, darf
noch nicht Auto fahren." :out :t2)

```

```

(((SENT-MARKER . ".")
  (:VF
    ((SEM (:HEAD "mann") (:MODS "klein") (:QUANTIFIER "d-det"))
      (:AGR
        ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :M)
          (:NUMBER . :S) (:CASE . :NOM)))
        (:END . 3) (:START . 0) (:TYPE . :NP))
      (:CONTENT

```

```

(:MF
  ((:SEM (:HEAD "zur") (:COMP (:HEAD "schule"))))
  (:SUB
    ((:SEM (:HEAD "schule"))
     (:AGR
      ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :F)
       (:NUMBER . :S) (:CASE . :AKK))
      ...
      ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :F)
       (:NUMBER . :S) (:CASE . :NOM)))
     (:END . 2) (:START . 1) (:TYPE . :NP)))
    (:AGR
     ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :F)
      (:NUMBER . :S) (:CASE . :DAT)))
     (:END . 2) (:START . 0) (:TYPE . :PP)))
  (:VERB (:LENGTH . 1) (:END . 8) (:START . 7)
  (:MORPHO-INFO
  (:AGR
   ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 2) (:GENDER . :NO)
    (:NUMBER . :P) (:CASE . :NO))
   ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3) (:GENDER . :NO)
    (:NUMBER . :S) (:CASE . :NO)))
   (:FINIT . T) (:GENUS . :AK) (:TENSE . :PRE))
   (:STEM . "geh") (:FORM . "geht") (:TYPE . :VERB))
  (:LENGTH . 3) (:TYPE . :SPANNSATZ))
  (:LENGTH . 5) (:END . 8)
  (:REL-PRON
  "der"
  ("d-det"
   (((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :M)
    (:NUMBER . :S) (:CASE . :NOM))
   ...
   ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
    (:NUMBER . :P) (:CASE . :GEN)))
   . :DEF)))
  (:START . 3) (:TYPE . :REL-CL))
  ("," ("," NIL . :INTP)))
  (:MF ("noch" ("noch" NIL . :PART)) ("nicht" ("nicht" NIL . :PART))
  ((:SEM (:HEAD "auto")))

```

```

(:AGR
  ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
   (:NUMBER . :S) (:CASE . :AKK))
  ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
   (:NUMBER . :S) (:CASE . :DAT))
  ((:TENSE . :NO) (:FORM . :NO) (:PERSON . 3) (:GENDER . :NT)
   (:NUMBER . :S) (:CASE . :NOM)))
(:END . 3) (:START . 2) (:TYPE . :NP)))
(:VERB (:LENGTH . 2) (:END2 . 14) (:START2 . 13) (:END . 10)
(:START . 9)
(:MORPHO-INFO
  (:AGR
    ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 3) (:GENDER . :NO)
     (:NUMBER . :S) (:CASE . :NO))
    ((:TENSE . :PRES) (:FORM . :FIN) (:PERSON . 1) (:GENDER . :NO)
     (:NUMBER . :S) (:CASE . :NO)))
    (:FINIT . T) (:GENUS . :AK) (:TENSE . :PRES))
  (:SCOPE (:LENGTH . 1) (:END . 14) (:START . 13)
  (:MORPHO-INFO
    (:AGR
      ((:TENSE . :SUBJUNCT-1) (:FORM . :FIN) (:PERSON . :ANREDE)
       (:GENDER . :NO) (:NUMBER . :P) (:CASE . :NO))
      ...
      (:GENDER . :NO) (:NUMBER . :S) (:CASE . :NO)))
    (:FINIT . :INF) (:GENUS . :AK) (:TENSE . :PRES))
    (:STEM . "fahr") (:FORM . "fahren") (:TYPE . :VERB))
    (:STEM . "duerf") (:FORM . "darf fahren") (:TYPE . :MODVERB))
    (:END . 15) (:START . 0) (:LENGTH . 14) (:TYPE . :KERNSATZ)))

```

7.4 Recognition of Grammatical Functions

Basic functions The clause together with its grammatical function can be determined via the following functions (where their “-HTML-” counterparts are described below):

1. (PARSE-GF-FROM-STRING <A TEXT STRING> :TIME <BOOLEAN>)
2. (PARSE-GF-FROM-FILE <A PATHNAME STRING> :TIME <BOOLEAN>)

They behave in the same way as the functions called for performing clause recognition (i.e., PARSE-FROM-STRING and PARSE-FROM-FILE the above).

HTML-based functions The following two functions map the resulting output to some HTML format and stores it into a file.

1. (PARSE-GF-HTML-STRING <A TEXT STRING> :TIME <BOOLEAN>)
2. (PARSE-GF-HTML-FILE <A PATHNAME STRING> :TIME <BOOLEAN>)

They behave in the same way as the HTML-based functions called for clause recognition (see above).

Some general remarks After clause recognition, the available information consists basically in a dependency tree provided with upper borders limiting the attachment possibilities of non-head elements, which we call *modifiers*.¹ As a further step in during chunk parsing the *grammatical function recognition module* (GFR) takes this kind of structures as its input, and computes an additional layer of information, consisting of:

1. The identification of possible *arguments* on the basis of the lexical subcategorization information available for the local head. We call the resulting structure a (partial) *underspecified functional description* (UFD).
2. The marking of the other non-head elements of the UFD as *adjuncts*, possibly by applying a distinctive criterion for standard and specialized adjuncts. Adjuncts —opposed to arguments, for which an attachment resolution is attempted— have to be considered underspecified wrt. attachment: in other words, their dependency relation to the head counts as an upper border rather than an attachment.

Currently, GFR applies only for verbal groups (no analysis of argumental structure internal to other classes of constituents, like NPs or ADJPs for

¹A terminological remark: here we use the label *modifier* only in connection to the status of these elements as non-heads. In other words, modifiers are not opposed to arguments: they include *both* candidate *arguments* and *adjuncts*, to be distinguished by the GFR module described here.

example, has been provided so far). Therefore the recursion potential is limited to subclauses (either as arguments or adjuncts), that are internally analyzed for GFs in the same way as the main clause is, while all the other constituents maintain the same structure they had in the input structure. For example, the result computed by GFR for the the sentence “Der Mann sieht die Frau mit dem Fernrohr”, i.e., its UFD is as follows:

```
(((:SYN
  (:SUBJ
    (:RANGE (:SEM (:HEAD "mann") (:QUANTIFIER "d-det"))
      (:AGR
        ((:PERSON . 3) (:GENDER . :M)
          (:NUMBER . :S) (:CASE . :NOM)))
        (:END . 2) (:START . 0) (:TYPE . :NP)))
    (:OBJ
      (:RANGE (:SEM (:HEAD "frau") (:QUANTIFIER "d-det"))
        (:AGR
          ((:PERSON . 3) (:GENDER . :F)
            (:NUMBER . :S) (:CASE . :NOM))
          ((:PERSON . 3) (:GENDER . :F)
            (:NUMBER . :S) (:CASE . :AKK)))
          (:END . 5) (:START . 3) (:TYPE . :NP)))
      (:NP-MODS)
      (:PP-MODS
        ((:SEM (:HEAD "mit")
          (:COMP (:QUANTIFIER "d-det") (:HEAD "fernrohr"))))
        (:SUB
          ((:SEM (:HEAD "fernrohr") (:QUANTIFIER "d-det"))
            (:AGR
              ((:PERSON . 3) (:GENDER . :NT) (:NUMBER . :S)
                (:CASE . :DAT)))
              (:END . 8) (:START . 6) (:TYPE . :NP)))
            (:AGR
              ((:PERSON . 3) (:GENDER . :NT)
                (:NUMBER . :S) (:CASE . :DAT)))
              (:END . 8) (:START . 5) (:TYPE . :PP)))
          (:SC-MODS)
```

```

(:PROCESS
  (:COMPACT-MORPH
    ((:TEMPUS . :PRAES) ... (:GENUS . :AKTIV)))
  (:MORPH-INFO
    ((:TENSE . :PRES) ... (:NUMBER . :S)
     (:CASE . :NO)))
  (:ART . :FIN) (:STEM . "seh") (:FORM . "sieht")
  (:TYPE . :VERBCOMPLEX))
(:FRAME ((:NP . :NOM) (:NP . :AKK)))
(:START . 0) (:END . 8)
(:SQL-TYPE . :GF-VERB-NODE)
(:TYPE . :SUBJ-OBJ)))

```

Subcategorized GFs The grammatical functions recognized by GFR correspond to a set of role labels, implicitly ordered according to an *obliquity hierarchy*, including:

- SUBJ: deep subject;
- OBJ: deep object;
- OBJ1: indirect object;
- P-OBJ: prepositional object;
- XCOMP: subcategorized subclause.

These labels are meant to denote *deep grammatical functions*, such that, for instance, the notion of subject and object does not necessarily correspond to the surface subject and direct object in the sentence. This is precisely the case for passive sentences, whose arguments are assigned the same roles as in the corresponding active sentence.

Adjuncts All the NPs, PPs and Subclauses that have not been recognized as arguments compatible with the selected frame are collected in *adjunct*

*lists*². We distinguish two different sets of adjunct lists; the first one includes three *general adjunct lists*, namely:

- NP-MODS, for modifier NPs;
- PP-MODS for modifier PPs;
- SC-MODS for modifier Subclauses.

After grammatical functions recognition, all the three above attributes are present (possibly with an empty list as value) in every UFD . Besides them, a second set of *special adjunct lists* is available. The most remarkable feature of *special adjunct lists* is that they are not defined as a fixed set, being rather triggered “on the fly” by the presence of some special information in adjunct phrases themselves: in the current implementation, this “special information” is encoded by means of the feature SUBTYPE. It is possible to define specialized finite state grammars (see??), aimed at identifying restricted classes of phrases (for instance, temporal or locational expressions), which impose for that phrases to be treated as special adjuncts by means of the value assigned to SUBTYPE. In the GFR grammar, a set of correspondences between values of SUBTYPE and special adjunct lists can be declared; here is some example:

- {LOC-PP, LOC-NP, RANGE-LOC-PP} \mapsto LOC-MODS
- {DATE-PP, DATE-NP} \mapsto DATE-MODS

This means that, for example, if a modifier marked as LOC-NP is present in an UFD, then an attribute LOC-MODS is going to be created in the corresponding UFD after GF recognition, whose value is a list containing that modifier; if another modifier belonging to the same class (say a RANGE-LOC-PP) is found, it is simply added to that list. The same mechanism applies for each class of SUBTYPE values considered in the GFR grammar.

²They are basically list-valued attribute-value pairs. In the actual format used in STP modules, they are expressed as *associative lists* of the form (label . (item1 item2 ...)) or the equivalent (label item1 item2 ...).

A EBNF syntax for output of text tokenizer

Note that all tokens between a left and right pointmark are collected together with the right pointmark into one list. However, since this is actually too simple, we actually flatten this embedded list representation in the next processing units. A note on the form of the BNF: only nonterminals are written using the italics font. Terminal characters are enclosed in single quotes ‘ ’, and strings are enclosed in duple quotes “ ”:

<i>Tokenstream</i>	::=	‘(’ <i>Tokenlist</i> * ‘)’
<i>Tokenlist</i>	::=	‘(’ [<i>Wordform</i> <i>SpecialWords</i> <i>Tokenassoc</i>]* <i>Pointmark</i> ‘)’
<i>Wordform</i>	::=	“ <i>Letter</i> * (‘-’ <i>Letter</i> ⁺)* “” <i>LetterLetter</i> *”
<i>SpecialWords</i>	::=	<i>Abbrev</i> <i>Clock</i> <i>Ct</i> <i>St</i>
<i>Tokenassoc</i>	::=	<i>Keywrd</i> <i>Quoted</i> <i>Integer</i> <i>Ordinal</i> <i>Time</i> <i>Date</i> <i>Cluster</i> <i>Special</i>
<i>Pointmark</i>	::=	“.” “!” “?” “&”

B Morphological features

The following table enumerates the features together with their domain:

:cat	→	:n :v :aux :modv :a :attr-a :def :indef :prep :relpron :perspron :refpron :posspron :whpron :ord :card :vpref :adv :whadv :coord :subord :intp :part
:mcat	→	:n-adj :det-word
:sym	→	:open-para :closed-para :!sign :questsign :separator :dot :dotdot :semikolon :comma
:comp	→	:p :c :s
:comp-f	→	:pred-used
:det	→	:none :indef :def
:tense	→	:pres :past :subjunct-1 :sibjunct-2
:form	→	:fin :infin :infin-zu :psp :prp :prp-zu :imp
:person	→	1 2 :2a 3 :anrede
:gender	→	:nfm :m :f :nt
:number	→	:s :p
:case	→	:nom :gen :dat :akk

C BNF for name subgrammars

Company names:

Firm-Expr: Firm-NP, Firm-PP

```
Firm-Expr -> [ start: number
                end: number
                agr: Morphix-Vector
                sem: [ loc: string
                      status: "i.L."
                      abbr: string
                      name: string
                      comp-form: CFs ] ]
```

```
Firm-NP -> [ sem: [ nation: string
                   business: string
                   daughter: string
                   mother: string
```

part-owner: string]]

Firm-PP -> [prep: string
sem: [nation: string
business: string
daughter: string
mother: string
part-owner: string]]

Stand-alone-names -> [Sem: [short-form: T|F]]

Firm-Coor-NP -> [start: number
end: number
conjuncts: list(Firm-Expr)]

CFs: "ag", "gmbh", "gruppe", "holding",
"trust", "corporation",
"aktiengesellschaft",
"Inc", "Corp", "Ltd",
"Limited", "Corp", "Incorporated"

Special year expression:

Year-Expr -> [start: number end: number
sem: [year1: number year2: number
months: [from: number to: number]
fiskaljahr: T|F geschjahr: T|F
geschper: T|F]]

D EBNF of current clause grammar

Verb-node	→	[:verb: Verbcomplex :NPs : list(NomObj) :PPs : list(PraePhr) :dir-speech : verb-structure[verb art fin] :subclauses : list(Subcl) :relative cl.: list(Relcl) :infinCompl : InfCompl :start: number :end: number]
Coord-verb-node	→	[:conj: und oder jedoch aber ... :art: :complete :incomplete :conjuncts: list(Verb-node)]
InfCompl	→	[:subconj: ohne anstatt um bare :content: verb-structure[:verb :art :infin]
Subcl	→	[: subconj: weil wenn als dass indem <i>dots</i> :content: verb-structure[:verb :art :infin]]
Relcl	→	[:rel-pron: [:form: der , die das <i>dots</i> :morph-info: Dnf-Vector] :content: verb-structure[:verb :art :infin]]
Verbcomplex	→	[:form: string :stem: string :pred-adj: T F :modal: string :negation: T F :art: :fin :infin :part morph-info: [tempus: Tempus :genus: :active :passive :person: 1 2 3 :numerus: :s :p modus: Ind Konj]]
Tempus:	→	:perf :pres :plusquamperf :fut1 :fut2 :imperfekt
NomObj:	→	NP Coord-NP Firm-NP PersPron Firm-Coor-NP
PraePhr:	→	PP Year-Expr Firm-PP

References

- [Finkler and Neumann, 1988] W. Finkler and G. Neumann. Morphix: A fast realization of a classification-based approach to morphology. In H. Trost, editor, *Proceedings der 4. Österreichischen Artificial-Intelligence Tagung, Wiener Workshop Wissensbasierte Sprachverarbeitung*, Berlin, August 1988. Springer.
- [Neumann and Mazzini, 1998] G. Neumann and G. Mazzini. *Domain-adaptive Information Extraction*. DFKI, technical report, 1998.
- [Neumann *et al.*, 1997] G. Neumann, R. Backofen, J. Baur, M. Becker, and C. Braun. An information extraction core system for real world german text processing. In *5th International Conference of Applied Natural Language*, pages 208–215, Washington, USA, March 1997.
- [Neumann *et al.*, 2000] G. Neumann, C. Braun, and J. Piskorski. A Divide-and-Conquer Strategy for Shallow Parsing of German Free Texts. In *ANLP-2000*, Seattle, Washington, pages 239-246, 2000.
- [Neumann and Piskorski, 2002] G. Neumann and J. Piskorski. *A Shallow Text Processing Core Engine*. In *Journal of Computational Intelligence*, Volume 18, Number 3, 2002, pages 451-476.