Eidgenössische Technische Hochschule Zürich
Ecole polytechnique fédérale de Zurich
Politecnico federale di Zurigo
Swiss Federal Institute of Technology Zurich

Department of Information Technology and Electrical Engineering
Computer Engineering and Networks Laboratory

Web-Interface for Natural Language Parsers

Andy Staudacher

**Advisors: Tobias Kaufmann
Thomas Ewender**

Summer Term 2007

Preface

For experts, a web interface to natural language parser can serve as a tool to get insights in the applied formal grammars. For laymen, it could serve as a demonstration tool for natural language parsing.

This report focuses on the design and implementation of such a web interface.

Common visualizations have been implemented using basic web technologies and an abstract parser interface has been defined such that the web interface can be reused with any natural language parser. All parser specific data and settings can be configured in a central place.

The report is split into the following chapters:

Chapter Introduction explains the task to fulfill.

Chapter Concepts presents the underlying concepts.

Chapter Design Space Exploration discusses alternative designs on a technical level.

Chapter Implementation describes the chosen system architecture and the software design in detail.

Chapter Conclusions and Outlook summarizes the report and gives an outlook on some possible improvements and extensions.

Author:	Andy Staudacher	andress@ee.ethz.ch
Advisors:	Tobias Kaufmann	kaufmann@tik.ee.ethz.ch
	Thomas Ewender	ewender@tik.ee.ethz.ch
Supervisor:	Beat Pfister	pfister@tik.ee.ethz.ch
Professor:	Lothar Thiele	thiele@tik.ee.ethz.ch

Acknowledgements

Thanks to my advisors, Tobias Kaufmann and Thomas Ewender, for their guidance, constant feedback, flexibility and support. I would further like to thank the Speech Processing group of the Computer Engineering and Networks Laboratory at ETH Zurich for the opportunity to realize this project.

Contents

List of Figures

Chapter 1

Introduction

The complexity of natural languages[1] and the processing thereof is mostly inconceivable for people not intimately familiar with natural language processing (NLP). A web-based interface to a natural language parser could convey a glimpse of this complexity to the interested mind.

Experts on the other hand benefit from such a tool by exploring the grammar of a natural language parser interactively instead of studying the theory behind it directly.

The subject of this report is the design and implementation of a web interface for natural language parsers. The focus is on application and user interface design while natural language parsing, specifically the grammar and its syntax, is defining the domain.

The resulting application supports the user in entering natural language, parses the text, lets the user browse the parse results and provides common visualizations thereof.

Chapter 2 discusses the task on a conception level, implementation alternatives are compared in chapter 3. Chapter 4 documents the chosen design and its implementation. Finally, the report is concluded in chapter 5.

In this chapter the requirements for the application are discussed in section 1.1 and a we take a look at related work in section 1.2.

1.1 Requirements

The design and implementation of a web interface for natural language parsers has several aspects and the overall solution consists of multiple components.

In the following we define the requirements for the application to be designed and implemented.

1.1.1 Development of a User Interface

The user interface should be accessible to laymen. It should be easy to understand and use without reading any documentation. On the other hand, experts are interested in efficiency and detailed information. The user interface should provide efficient access to key properties of the parse results and should visualize them.

[1]Natural languages are languages spoken and written by humans for general communication. As opposed to formal languages like programming languages or languages to describe logical or mathematical formalisms.

The requirements for the user interface are described in the following:

Input Process

The user should be supported in entering sentences that can be processed by the natural language parser. A common constraint is that a parser has a limited lexicon. There should be a way to guide the user to use the available words.

Furthermore, a collection of documented examples could be used to demonstrate interesting natural language phenomena.

Exploring Ambiguity

Sentences in natural language are often ambiguous, allowing for multiple readings (interpretations). Natural language parsers generally detect such ambiguities and return multiple parse results reflecting the existence of multiple readings.

The user interface should support the user in two tasks: In quickly finding and picking the right reading and it should also reveal the different sources of ambiguity in a concise manner.

Visualization of the Results

A specific reading should be visualized in a way that is common in the field of natural language parsing, specifically with a derivation tree (see section 2.1.3).

It should be possible to annotate the visualization with notes and references to explanatory publications.

1.1.2 Application Development

The application design should follow established design patterns and make use of existing components where possible.

Parser Interface

The interface to the parser should be abstract and general enough such that the web interface can be used for different grammars and parsers.

Implementation

The web interface should be as portable as possible and the deployment should be simple and well documented.

Documentation

The application should be well documented on a technical level such that it can be maintained and extended easily.

1.2 Related Work

While there are several existing natural language parser frameworks, none of them seems to include a web interface that meets the specified requirements.

1.2.1 Heart Of Gold

Heart of Gold (HoG) is “an XML-based middleware for the integration of deep and shallow natural language processing components” [1]. HoG meets several of our requirements: It has an abstract interface to natural language parsers, it provides an application interface to build user interfaces upon and it has a clean and well documented application design. While it falls short when it comes to meeting our user interface requirements, it could serve as a basis for user interface that meets our purpose.

The reason why HoG was not chosen as the framework for this web interface is that HoG’s focus on integration of deep and shallow parsers is quite different from ours. The consequence of this is that the application is designed to solve another problem and that it would have to be changed substantially to serve for our purposes. The costs involved in learning and refactoring an existing framework have been considered to outweight the advantages of using an existing framework.

Nevertheless, HoG can still serves as an example in the design phase of our own solution.

1.2.2 LinGO English Resource Grammar On-Line

While there is a on-line version of the LinGO English Resource Grammar (ERG), its sources are not included in the public ERG package. Furthermore, it does not focus on revealing ambiguities and the visualizations of the results are rather primitive to what we have in mind.

1.2.3 The Babel-System: HPSG Interactive

The web interface for the Babel-System meets several of our goals but only the Babel-System itself is publicly available, excluding the web interface. Also, it does not offer any means to browse the parse results effectively and thus does not support the user in resolving the ambiguities.

1.2.4 FreeLing

FreeLing is an “Open-Source Suite of Language Analyzers” and includes a web interface. It provides a sophisticated visualization of derivation trees, but it does not work on most web browsers without installing a browser plugin. And it lacks of support to browse and explore whole sets of results. // The application itself is trimmed towards a different goal and does not meet our requirements.

1.2.5 Alpino

Alpino is a “dependency parser for Dutch”. It features a similar web interface as FreeLing and the Babel-System, but the web interface is not publicly available.

Chapter 2

Concepts

In this chapter we will look at the key concepts of the intended web interface for natural language parsers. After looking at linguistic concepts in section 2.1, we will examine the user interaction process in section 2.2 and close with the discussion of the application interface to the natural language parser in section 4.6.

2.1 Linguistic Concepts

To understand the work flow and the components that are involved in the design of a web interface for natural language parsers, some linguistic concepts need to be introduced.

Natural language parsing is a broad field in itself. Our focus lies on the grammar of natural language and its syntax.

The following list of concepts is kept to a minimum to support the understanding of the application design process.

For a good yet concise introduction to key concepts of natural language parsing and syntax theory in particular read [2, Sag et al. - Syntax Theory: A Formal Introduction].

2.1.1 Natural Language Parsing

Natural language parsing is concerned with the structural analysis of natural language. Given a formal gramar, a parser tries to find sequences of rule applications (derivation sequences) which produce a given sentences. For grammars with a context-free skeleton, a derivation sequence can be represented as a tree.

2.1.2 Parts Of Speech

Part of speech classes represent words which have common syntactic properties.

Example: When parsing the English sentence “the dog ate the bone” the words are tagged: “the” - *determiner*, “dog” - *noun*, “ate” - *verb*, “the” - *determiner*, and “bone” - *noun*.

Depending on the parser, different sets of parts of speech are identified. A classification which works for our application is the following:

- verb : (“schlafen”)
- noun : common nouns (“Buch”), proper nouns (“Herbert”), pronouns (“er”)

- determiner : determiners (“ein [Mann]”) including attributive pronouns (“ein [Mann]”)
- preposition : preposition (“mit [dem Löffel]“) including pronominal-adverbials (“damit”)
- conjunction : (“weil [es gegnet]“)
- adjective : attributive and predicative adjectives (“kleine [Kinder]”, “[das Kind ist] klein”) and nominalized adjectives (“[die] Kleinen”)
- adverb : any adverbials which do not fall into some other category

2.1.3 Derivation Tree

Derivation trees (or syntax trees) are a common visualization of parse results for grammars with a context-free skeleton. Figure 2.1 shows an example for a derivation tree representing the parse result from parsing the English sentence “the dog ate the bone”.

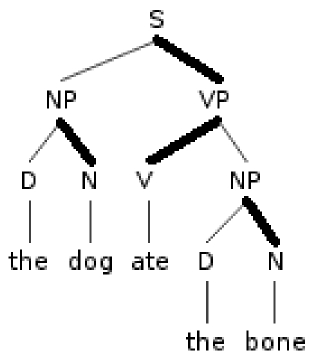


Figure 2.1: Derivation tree for the English sentence “the dog ate the bone”

One of the key properties visualized in a derivation tree is the part of speech for each word. In this example the parts of speech are illustrated as the direct parents of word (leaf) nodes. D stands for determiner, N for noun, and V for verb.

We will come back to this example and use figure 2.1 in the following to introduce further key concepts.

2.1.4 Grammar Rules

One aspect hidden in derivation trees is the *grammar rules* that have been applied to derive the tree.

The term “context-free skeleton” refers to one of the principles of a class of grammars. A mimimal set of production rules to generate the above example is illustrated in table 2.1.

Each production rule substitutes the non-terminal symbol on the left with the symbols on the right of the arrow operator. It is called context-free because the left-hand-side of the production rule consists of a single non-terminal symbol. As the name implies, a terminal symbol marks the end of a substitution sequence and cannot be substituted with any other symbol.

S	$\rightarrow$	NP	VP
NP	$\rightarrow$	D	N
VP	$\rightarrow$	V	NP

Table 2.1: Production rules of a context-free grammar.

It is interesting to note that in the context of derivation trees, part of speach tags are often called "pre-terminals" (terminals are the leaves, non-terminals the inner nodes).

In figure 2.1 all tree nodes from the top (root node) to the actual words (leafs) are representing grammar rules. The sentence (S) consists of a noun phrase (NP) followed by a verb phrase (VP). The NP consists of a determiner (D, "the") followed by a noun (N, "dog"). The VP has a verb (V, "ate") and ends with a NP which consists again of a determiner (D, "the") and a noun (N, "bone").

Looking at it from another perspective, each node is an *argument* to the production rule of its parent (with the exception of the root node). To form an NP phrase, you need a determiner argument and a noun argument. The two *daughters* of a NP node represent its arguments.

Phrasal vs. Lexical Rules

The derivation tree from figure 2.1 only shows the phrasal level. Table 2.1 shows a list of *phrasal rules* that are used to derive the derivation tree.

The tree could be expanded to show morphological information, i.e. how the words are derived from the grammar's lexicon. Similarly to how phrasal rules define how parts of the whole phrase can be combined, *lexical rules*, like conjugation and declination, define how to derive words from lexical entries.

One differentiates between phrasal and lexical rules and between the phrasal and lexical level when discussing parse results and the corresponding derivation trees.

2.1.5 Phrases

On the phrasal level, each inner node represents a *phrase*, a subpart of the whole sentence.

Example: In the example of figure 2.1, the left noun phrase (NP) consists of the subpart "the dog" of the sentence.

Each phrasal node in a derivation tree thus not only implies that a phrasal rule has been applied, it also stands for a phrase.

2.1.6 Heads

Another concept illustrated in derivation trees are heads. Heads largely determine the syntactic properties of the phrase that they are part of. For instance, the noun in a noun phrase gives the rule its name and determines the properties of the noun phrase. In figure 2.1, a head of a phrasal rule is illustrated by having a thicker line to its parent than other arguments have.

Since heads largely determine the properties of a phrase, they also provide further constraints on what phrases can be combined to form a phrase.

2.1.7 Ambiguity

The syntax of natural language generally allows for multiple readings and thereby inherently leads to ambiguity.

Example: The German sentence “dass sie die Männer mit dem Feuerstuhl gejagt haben” illustrates some classes of ambiguity:

- Is “mit dem Feuerstuhl” associated with the subject “sie” or with the accusative “Männer”? Note: This sort of ambiguity is also called “PP-attachment ambiguity” (PP=prepositional phrase).
- Is “Männer” the plural form of “Mann” or the surname of “Wolfgang Männer”, a funeral director from Ingolstadt?
- Is “die” a determiner or a pronoun?
- Is “gejagt” a verb or an adjectival participle?

Depending on the sentence and how the ambiguities can be combined, several hundreds of readings can result. The author of the sentence may or may not be aware of the ambiguity and the reader may or may not resolve ambiguities from the context.

2.2 User Interface Design

Now that the key concepts of natural language parsing have been introduced, the processes in which users interact with the natural language parser can be explored and the user interface design can be discussed.

2.2.1 User Interface Guidelines

The following guidelines form the basic principles of the user interface design for our web interface:

- The design should be minimal eliminating all possible sources for distraction.
- The user interface should be concise and self-explanatory.
- The terminology and the graphical representations should conform with the domain-specific language and visualizations.
- Provide easy access to additional information for experts.
- Contextual help should be provided as necessary.
- The user interface should be consistent across all areas of the web interface.

2.2.2 User Interaction Process

The basic user interaction process can be divided into three parts:

1. Input Process - Entering a sentence and submitting a parse request.
2. Disambiguation - Browsing all resulting readings. Discovering ambiguities and finding the intended result.
3. Result Inspection - Viewing a specific reading with detailed information.

Each sub-process will be discussed with all associated components in the following sections.

2.2.3 Input Process

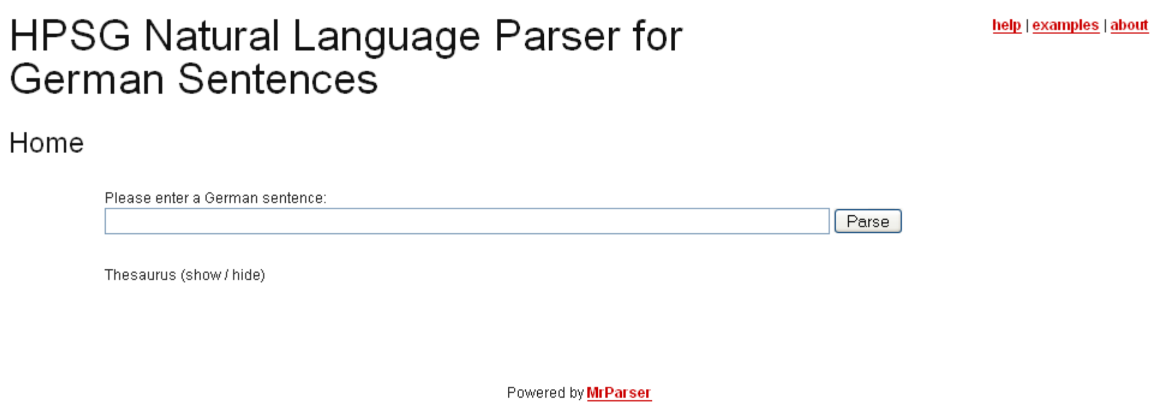
In the input process we are facing the challenge to guide the user to enter a sentence that can be parsed by the natural language parser.

The user is constrained by the selection of words since the lexicon of natural language parser is generally limited to a small to medium subset of the vocabulary that is used in every day communication.

Depending on the parser the user might also be constrained in the characters that are accepted by the system. A parser might not allow for quotes or punctuation marks. Or the parser might only handle lower-case letters of the English language and not accept digits at all.

For this purpose a number of measures are set into place to limit the user in actions that cannot be handled by the parser and to guide the user to enter a sentence that can be parsed by the parser.

The complete input page is illustrated in figure 2.2. The following sections discuss the individual components of the input page.


HPSG Natural Language Parser for German Sentences

help | examples | about

Home

Please enter a German sentence:

Parse

Thesaurus (show / hide)

Powered by MrParser

Figure 2.2: The input page with the input form and supporting elements.

Input Form

The input form to enter a sentence is kept minimal, resembling the interface of popular search engines. Figure 2.3 shows a small prompt specifying what to enter into the input form alongside the actual input element and a button to submit the parse request.

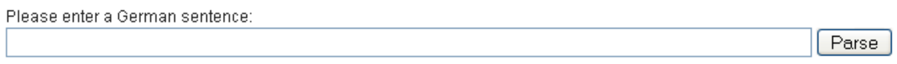


Figure 2.3: The minimal input form.

The input form has several constraints on the input that it allows. By adding behavior to the input form, the user is supported in the process of entering sentences that can be parsed.

Ignoring Unknown Characters: If the user types a character that is not allowed by the underlying parser, the keystroke is ignored. The user should thus quickly learn what characters are allowed and the user is trained to work around this limitation by choosing appropriate expressions.

Auto Completion: While entering text, the input form suggests alternatives to complete the word that is currently being typed. This is illustrated in figure 2.4. The user can then either continue to type the word to the end or the user can scroll through the suggested auto completions and select one of them.

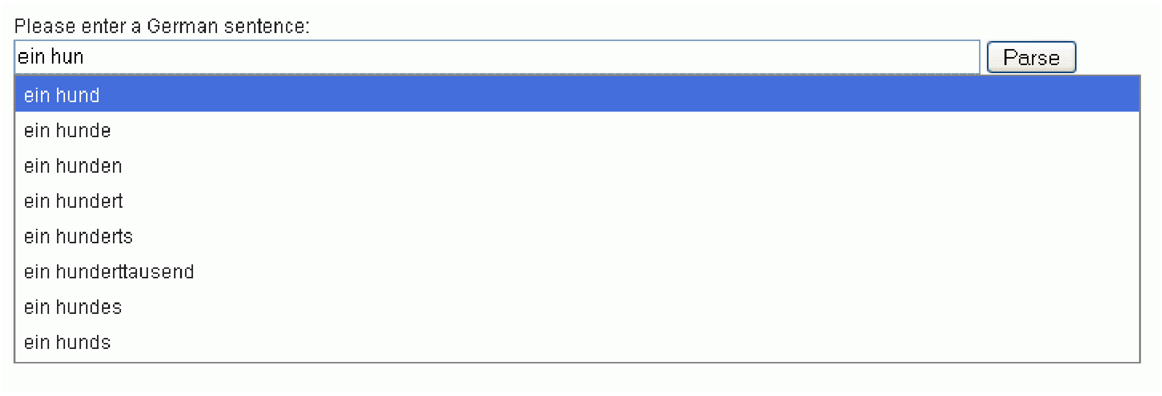
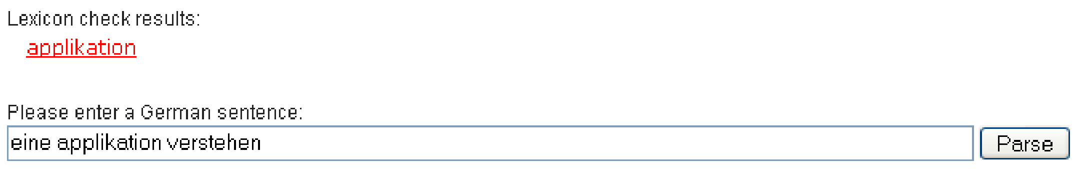


Figure 2.4: Auto completion behavior of the input form.

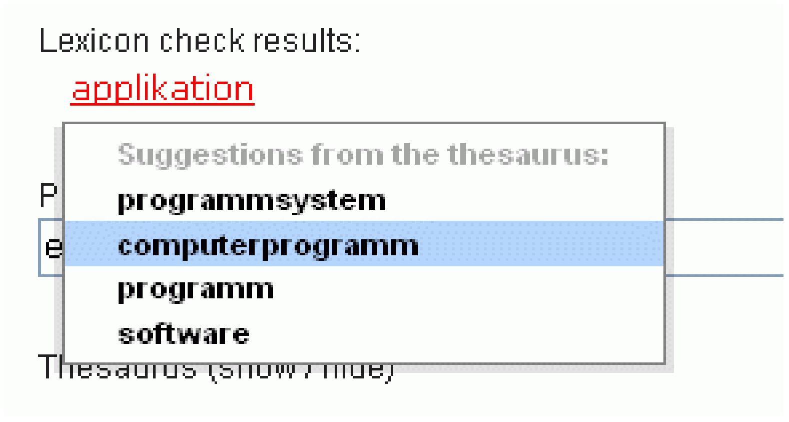
Lexicon Checks: Since the user may enter words that are unknown to the parser, the user is given immediate feedback on unknown words. While the user is typing the sentence, the system checks all words in the input form against the parser's lexicon.

Figure 2.5 shows how an unknown word is marked. The word is formatted like a hyperlink prompting the user to click on it. Figure 2.6 depicts the menu shown when the user clicks on an unknown word. It lists suggestions from the thesaurus for the unknown word. Once


Lexicon check results:
applikation

Please enter a German sentence:
eine applikation verstehen

Figure 2.5: An unknown word is reported by a lexical check.


Lexicon check results:
applikation

Suggestions from the thesaurus:
programmsystem
computerprogramm
programm
software

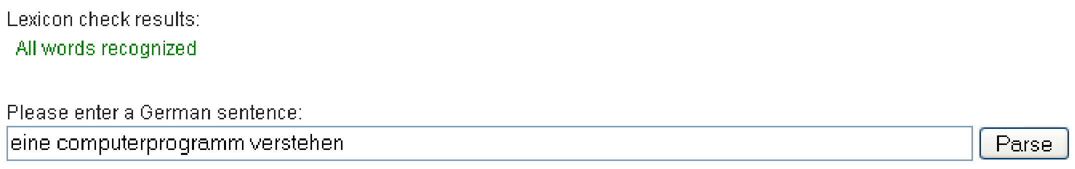
Figure 2.6: Suggestions from the thesaurus for an unknown word.

the user makes the choice for an alternative and clicks on a specific suggestion, the word is replaced in the input form as illustrated in figure 2.7.

It is worthwhile to note that there might not be a finite lexicon. This is the case with our system. A morphological analysis component may derive the syntactic properties of infinitely many words (most notably in German: noun-noun compound such as "Kraftfahrthaftpflichtschutz"). This is important as it justifies our decision to do the lexicon check by calling the parsing system (rather than employing a generated list of lexicon words).

Thesaurus

The thesaurus service that is responsible for providing suggestions for unknown words in the interface of lexicon check results is also exposed as a standalone component allowing the user to explore and find words that can be used to form a sentence. The user interface is shown


Lexicon check results:
All words recognized

Please enter a German sentence:
eine computerprogramm verstehen

Figure 2.7: A positive lexicon check result.

in figure 2.8.

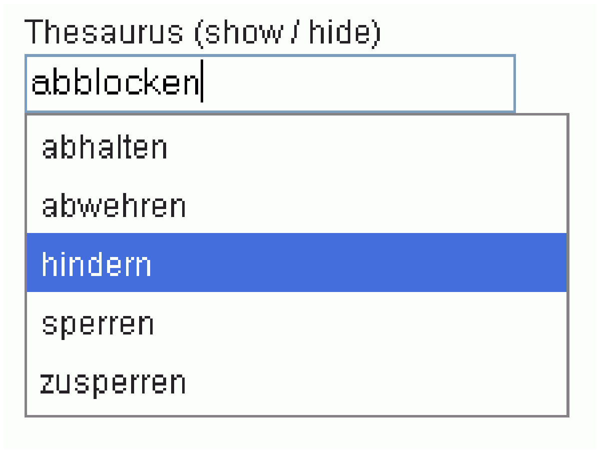


Figure 2.8: Thesaurus component showing several results for the given term.

Examples

An alternative input method is choosing a sentence from an existing collection of examples. A link in the upper right corner of the input page takes the user to the examples page, shown in figure 2.9.

The examples page shows a collection of sentences demonstrating natural language phenomena that are covered by the parser.

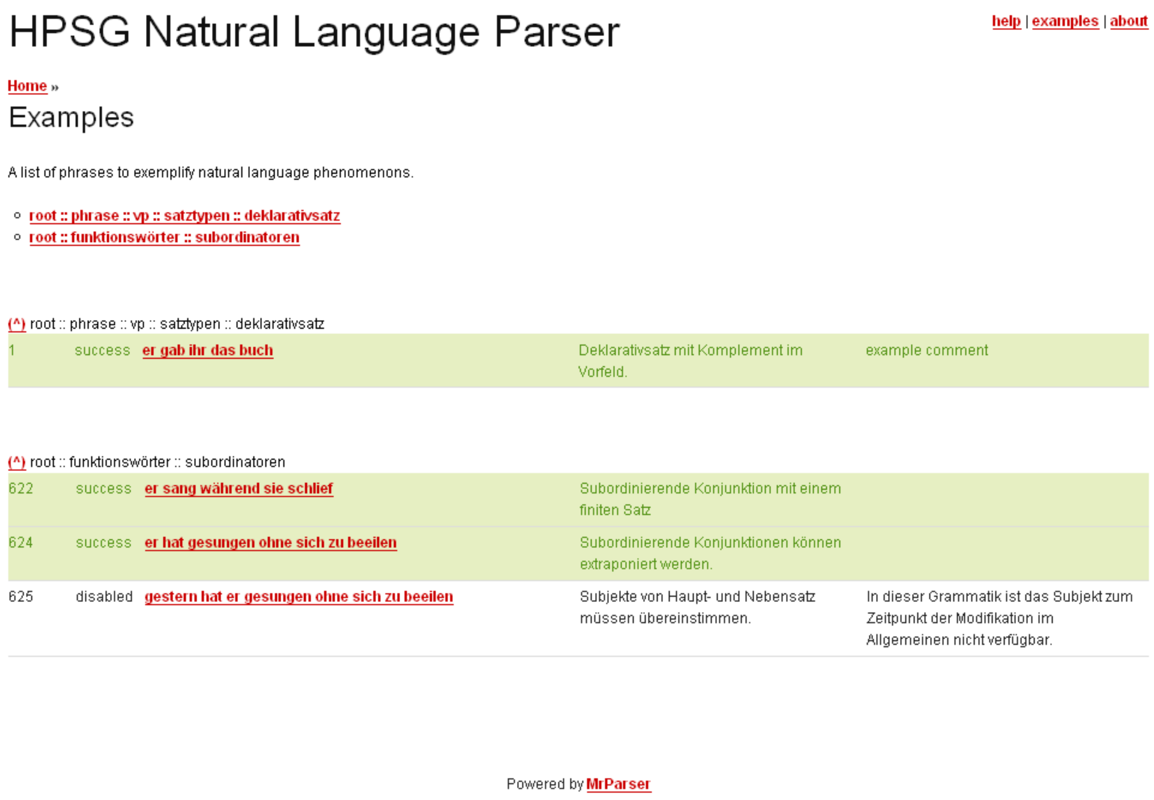
Process

The actual input process can be described as an interaction process with the above components. The user has three different ways to specify a sentence:

- The user can follow the navigation link to the examples page and clicks there on a specific sentence.
- The user can copy and paste a sentence into the input form and submit the parse request.
- The user can type a sentence manually into the input form and then submit the request.

While using the input form, the user is supported by periodic lexicon checks of the text within the input form and by auto completion suggestions. And as needed, the user can use the thesaurus form on the same page to find words that can be used in the input form.

Once the parse request has been submitted, the user is taken to the parse results page. There are either no possible readings at all, there might be an error because an unknown word has been entered or there might be one or more readings.



HPSG Natural Language Parser

help | examples | about

Home »

Examples

A list of phrases to exemplify natural language phenomenons.

- root :: phrase :: vp :: satztypen :: deklarativsatz
- root :: funktionswörter :: subordinatoren

(^) root :: phrase :: vp :: satztypen :: deklarativsatz

1	success	er gab ihr das buch	Deklarativsatz mit Komplement im Vorfeld.	example comment

(^) root :: funktionswörter :: subordinatoren

622	success	er sang während sie schlief	Subordinierende Konjunktion mit einem finiten Satz	
624	success	er hat gesungen ohne sich zu beeilen	Subordinierende Konjunktionen können extraponiert werden.	
625	disabled	gestern hat er gesungen ohne sich zu beeilen	Subjekte von Haupt- und Nebensatz müssen übereinstimmen.	In dieser Grammatik ist das Subjekt zum Zeitpunkt der Modifikation im Allgemeinen nicht verfügbar.

Powered by MrParser

Figure 2.9: The examples page.

2.2.4 Disambiguation

The focus on the parse results page (or parse browser) is to discover and resolve ambiguities and to find the intended reading. The process of resolving ambiguities is called *disambiguation*.

Figure 2.10 shows the parse browser showing that the parser found just two different readings for the specified sentence.

The parse browser consists of three components:

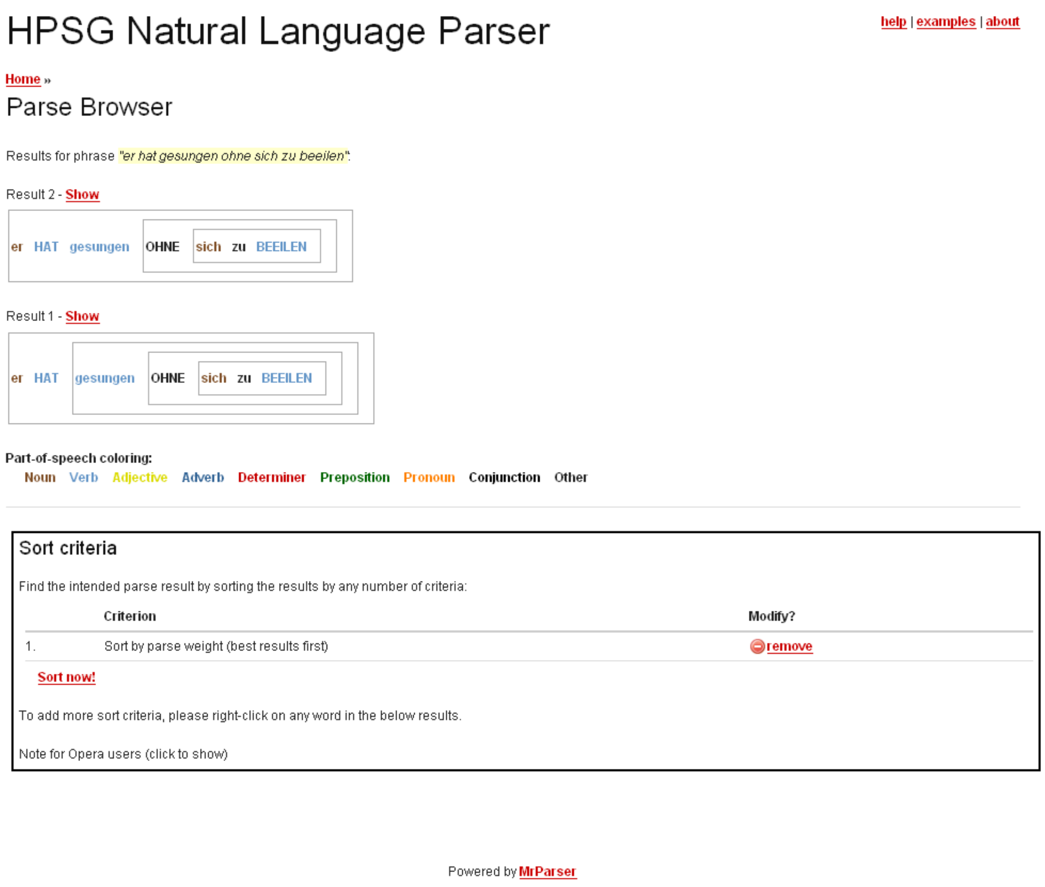
- A list of all possible readings (results) in a concise style to reveal sources of ambiguation and visualize key properties.
- A user interface to sort the result set according to several criterias.
- A color coded legend for the parts of speech.

Disambiguation Widget

The *disambiguation widget* is the concise visualization of a single reading revealing key properties on the first look and revealing sources of ambiguity when compared to readings in the same set of results.

Figure 2.11 shows a reading emphasizing the structure of the sentence by highlighting a specific phrase.

The reading also shows the parts of speech of all words on first sight.


HPSG Natural Language Parser

help | examples | about

Home »

Parse Browser

Results for phrase "er hat gesungen ohne sich zu beeilen".

Result 2 - Show

er HAT gesungen OHNE sich zu BEEILEN

Result 1 - Show

er HAT gesungen OHNE sich zu BEEILEN

Part-of-speech coloring:
Noun Verb Adjective Adverb Determiner Preposition Pronoun Conjunction Other

Sort criteria

Find the intended parse result by sorting the results by any number of criteria:

	Criterion	Modify?
1.	Sort by parse weight (best results first)	remove

Sort now!

To add more sort criteria, please right-click on any word in the below results.

Note for Opera users (click to show)

Powered by MrParser

Figure 2.10: Parse browser page showing two readings.

Visualizing the phrase structure: *Inclusion trees* were chosen as the visualization method of the disambiguation widget. Their compactness and the ability to fit the drawing into a prescribed area [3].

Other visualization aspects were easy to integrate into inclusion tree drawings. Parts of speech are color-coded and phrasal heads are emphasized by capital letters.

Figure 2.10 shows two different readings that only differ in their structure. The compact visualizations reveal this difference instantly and the phrase structure is further highlighted while checking a specific reading by hovering over the phrases.

A hyperlink next to the widget allows to inspect a specific reading in detail.

Sorting the Results

To support the user in finding the intended reading, one can sort the results by their key properties.

Sorting by phrase structure allows to sort all readings by a specific phrase. The user picks a specific word (see figure 2.12) and it sorts by the largest phrase of which the given word is the head. It lists readings with the longest matched phrase first and thus allows for quickly ruling out results that do not meet the intended phrase structure.

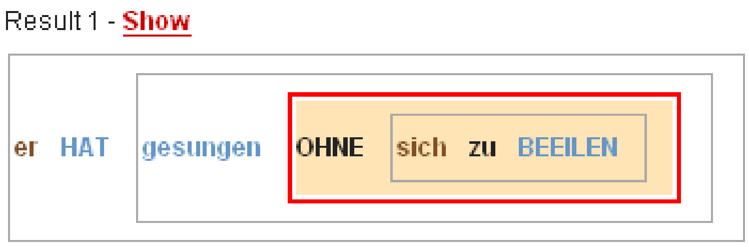


Figure 2.11: Disambiguation widget highlighting the phrase structure.

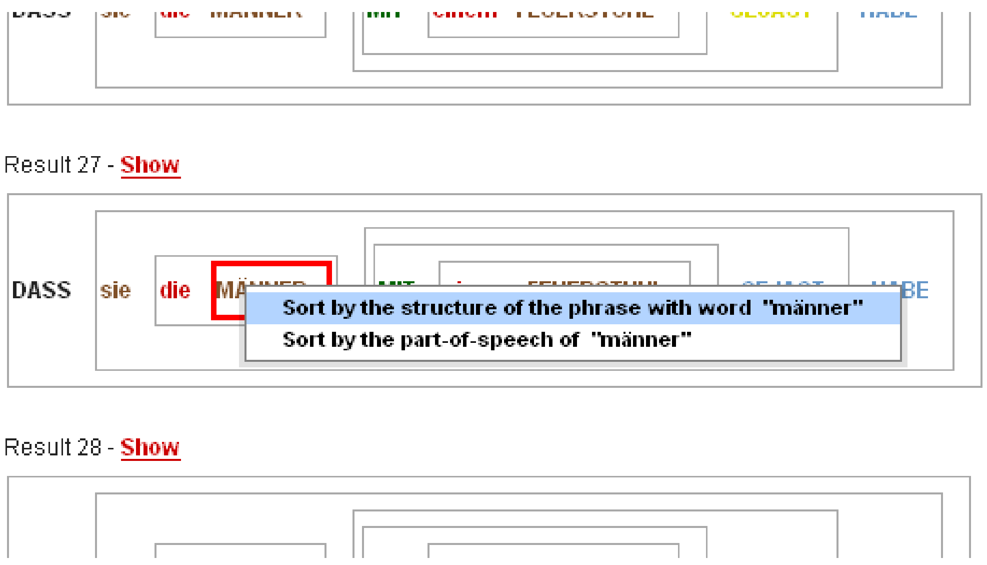

Result 27 - Show
DASS sie die MÄ
BE
Sort by the structure of the phrase with word "männer"
Sort by the part-of-speech of "männer"
Result 28 - Show

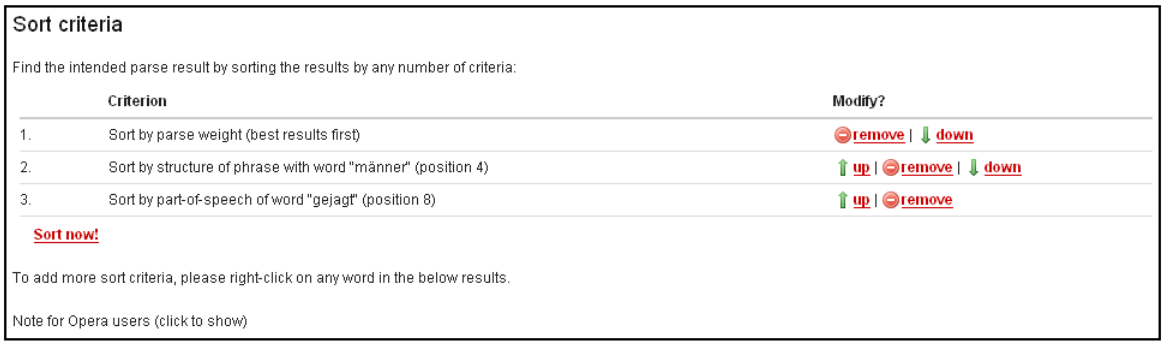
Figure 2.12: Adding a sort criterion.

Sorting by part of speech allows to sort all readings by the part of speech of a specific word. The user chooses a word whose part of speech should be compared and the criterion is added. Thus allowing e.g. to separate readings with “gejagt” as verb quickly from the same word as adjective.

Sorting by word features allows to sort all readings by a specific feature provided by the parser like genus or numerus. The user again picks a specific word, chooses one of the available features to compare by and adds the criterion to the list.

Sorting by weight allows to give control over the ordering to the parser. The parser can assign a weight to each reading to indicate what it considers a good or a bad match. This is also the default sort criterion.

Sort Interface The user interface for sorting the readings consists of multiple components. There is the main sort interface (see figure 2.13) with a list of active sort criteria and with options to manipulate the list.



Sort criteria

Find the intended parse result by sorting the results by any number of criteria:

	Criterion	Modify?
1.	Sort by parse weight (best results first)	remove | down
2.	Sort by structure of phrase with word "männer" (position 4)	up | remove | down
3.	Sort by part-of-speech of word "gejagt" (position 8)	up | remove

Sort now!

To add more sort criteria, please right-click on any word in the below results.

Note for Opera users (click to show)

Figure 2.13: Sort interface.

Also, each of the disambiguation widget instances serves as a source for sort criteria. By opening the context menu over a reading on the parse browser page a menu is opened to add more sort criteria (see figure 2.12).

Legend for color coding the parts of speech

The last element of the parse browser is a legend documenting the color coding of parts of speech as shown in figure 2.14

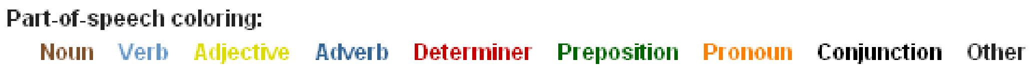


Figure 2.14: Color coding of parts of speech.

Process

The disambiguation process can be very short or a long discovery with multiple steps. Depending on the parse results, the result set can be as small as zero or a single reading or have as many as several hundreds of readings.

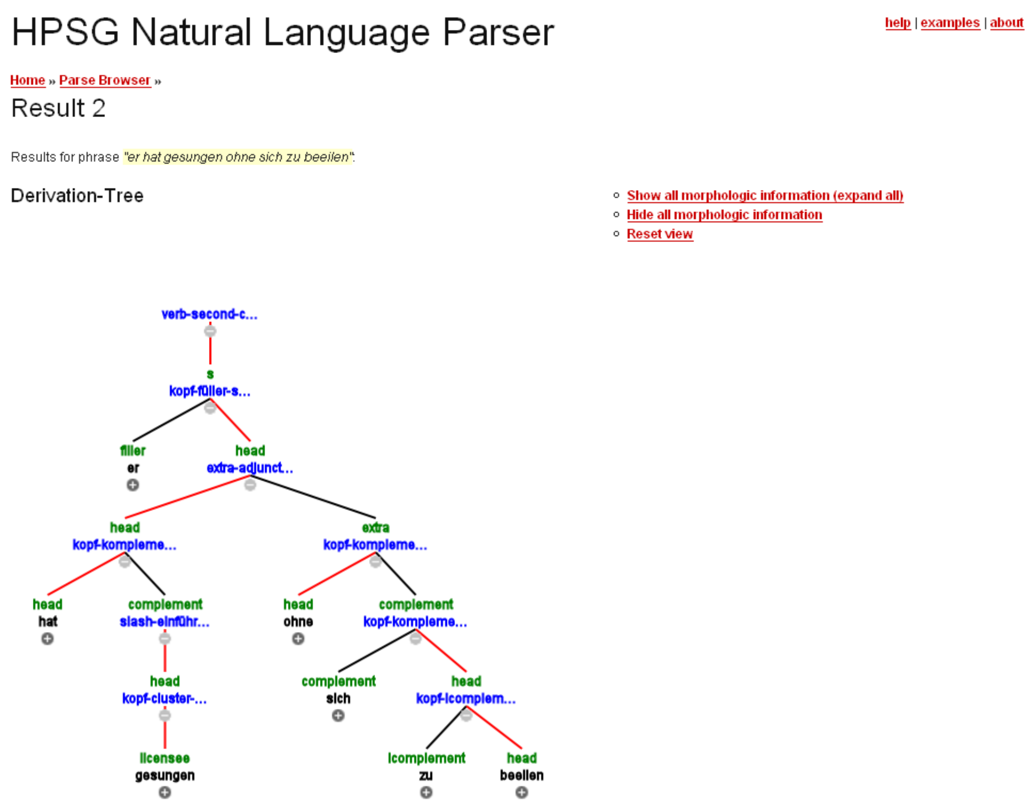
If there is only a single reading, the user might quickly take notice of the part of speech coloring and of the phrasal structure and then continue the inspection by checking the result in a detailed view.

If there are multiple readings, the user might be surprised by the ambiguities and compare the readings.

When there are dozens or even hundreds of readings it makes sense to sort them, adding more and more sorting criteria and to switch the order of the criteria to discover different phenomena and to find the intended result.

2.2.5 Result Inspection

The final step in the overall process is the inspection of a specific reading to get detailed information about a specific parse result. Figure 2.15 depicts the result viewing page which mainly consists of an interactive derivation tree widget.



HPSG Natural Language Parser

help | examples | about

Home » Parse Browser »

Result 2

Results for phrase "er hat gesungen ohne sich zu beeilen".

Derivation-Tree

- Show all morphologic information (expand all)
- Hide all morphologic information
- Reset view

Figure 2.15: Result view page.

Derivation tree widget

A derivation tree is used to provide a common visualization of natural language parse results. As illustrated in figure 2.16, it shows the name of the grammar rule that has been applied and the function under which that node acts as the argument to its parent in every tree node.

Head daughters are marked with a red link and for lexical nodes it also shows the text that they are representing.

This static behavior is augmented with interactive elements.

- By clicking on an icon under each node one can expand and collapse the sub tree of all nodes under that specific node.
- By clicking on a rule name further information about the rule is superimposed.
- Additional menu options allow to show / hide morphological information and to reset the view.

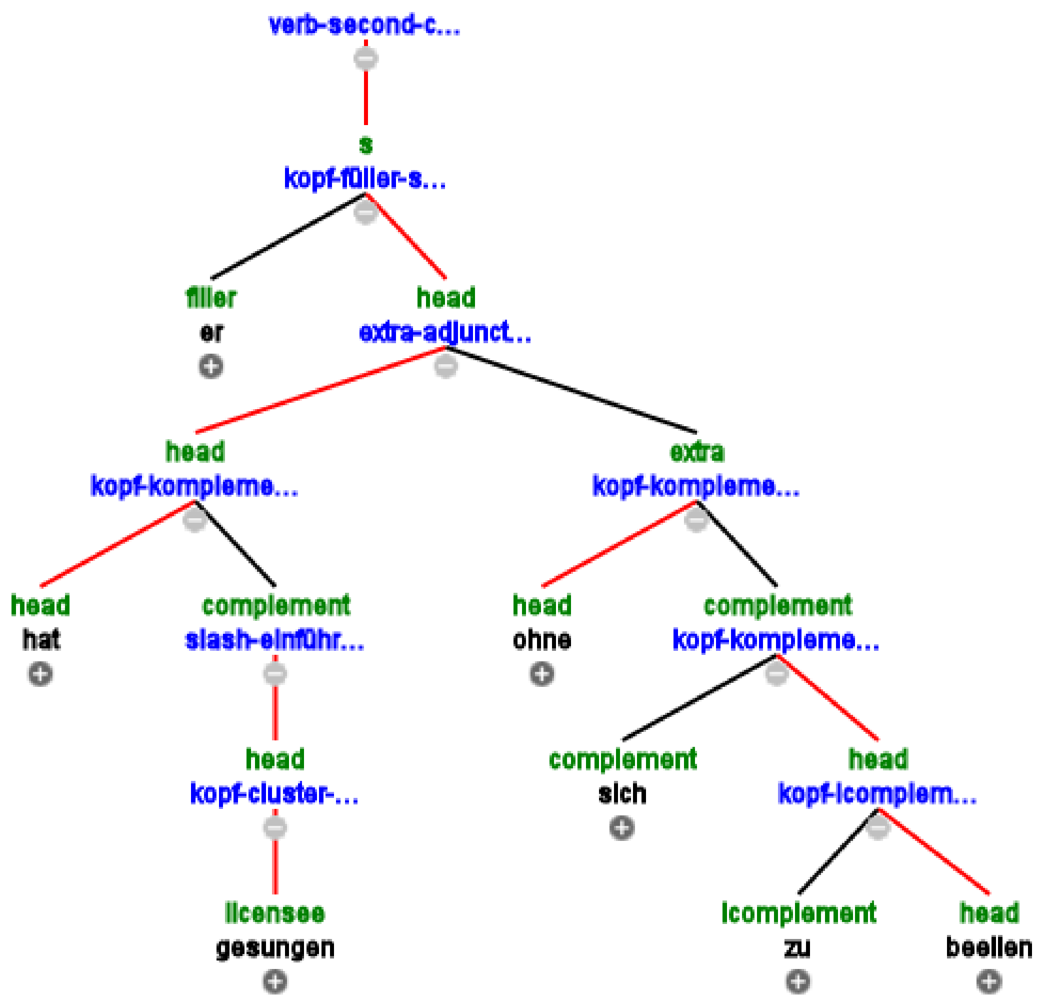


Figure 2.16: Derivation tree widget.

Process

When the user has reached the result viewing page the derivation tree is usually inspected in detail. A first look gives a good overview over the reading and its structure. By inspecting phrasal nodes and their rules a user might discover phenomena or verify assumptions.

The superimposed information about phrasal rules might feature hyperlinks to further reading as related publications.

The user might open multiple result detail views for different readings and switch windows to compare them.

Once the inspection process is over, the user might either close the window, or the user might return to either the parse browser or back to the input page directly.

2.3 Parser Application Interface

A key requirement of the web application is the modular design such that the web interface can be used with different grammars and parsers.

A detailed documentation of the data and file structures and of the contract between the parser and the web application can be found in section 4.6. In the following we give a brief overview of the parser specific data and the parser interface.

2.3.1 Request Handling

XML has been chosen as a simple message format between the web application and the parser.

One has to differentiate between lexicon check requests and parse requests.

A request consists of the method name that should be invoked (either lexicon check or parse) and the text to be parsed.

Parse Requests

The response for a parse request might look like:

```
<?xml version="1.0"?>
<!DOCTYPE parses SYSTEM "parses.dtd">
<parses>
  <parse weight="0.3">
    <node text=" der mann geht" ruleId="rule79">
      <node text="der mann" ruleId="rule3" ruleArgumentId="argument79.0">
        <node text="der" ruleId="rule13" ruleArgumentId="argument3.0"
              partOfSpeech="determiner"/>
        <node text="mann" ruleId="rule17" ruleArgumentId="argument3.1"
              partOfSpeech="noun"/>
      </node>
      <node text="geht" ruleId="rule3" ruleArgumentId="argument79.1"
            partOfSpeech="verb" featuresId="3"/>
    </node>
    <features featuresId="3">
      ...
    </featurea>
  </parse>
  <parse weight="0.6">
    ...
  </parse>
  ...
</parses>
```

The examples does not include any morphological information (word nodes have no children) for brevity. Generally, a parse result consists of a list of parses and each parse (reading) consists of a node tree representing the syntax tree (derivation tree) and with optional references to additional node features.

Lexicon Check Requests

A response for a lexicon check request specifies a list of unknown words. An empty list implies that all words are known to the parser.

An lexicon check response might look like:

```
<?xml version="1.0"?>
<!DOCTYPE parses SYSTEM "parses.dtd">
<status>
  <unknownWordException word="motorrad"/>
  <unknownWordException word="verfolgt"/>
</status>
```

2.3.2 Grammar

Since the grammar is static, it is not part of the parse request handling and needs to be specified separately.

The grammar is a list of phrasal and lexical rules. The rules and rule arguments can be annotated with a additional, unstructured information which is shown in the detailed result view.

A short excerpt of a grammar file might look like:

```
<?xml version="1.0"?>
<!DOCTYPE grammar SYSTEM "grammar.dtd">
<grammar>
  <rule ruleId="rule66" headIndex="1" type="phrasal">
    <ruleArgument ruleArgumentId="argument66.0">
      <annotation name="complement"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument66.1">
      <annotation name="head"/>
    </ruleArgument>
    <annotation name="kopf-komplement-schema-verbal"/>
  </rule>
  <rule ruleId="rule5" type="lexical">
    <ruleArgument ruleArgumentId="argument5.0">
      <annotation name="1"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument5.1">
      <annotation name="2"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument5.2">
      <annotation name="3"/>
    </ruleArgument>
    <annotation name="noun-noun-compound"/>
  </rule>
  ...
</grammar>
```

2.3.3 Thesaurus Data

The thesaurus service of the web interface depends on grammar-specific thesaurus data. Only thesaurus entries that map a term to words that are known to the parser are in this special, filtered thesaurus.

The parser needs to provide a text file with the thesaurus already filtered to match its own grammar.

2.3.4 Full Forms Lexicon

To provide the auto completion service during the input process, the web interface needs access to a lexicon of full forms, i.e. an exhaustive list of words in all their declinations, conjugations, etc.

This data is also static and needs to be provided when configuring the web interface.

2.3.5 List of Legal Characters

The last bit of information required by the web interface is an exhaustive list of legal characters, characters that can be parsed by the parser.

This list of characters can be used in the user interface to prevent the user from entering forbidden characters and it can be used in the web application to filter any malicious or illegal input before requesting the parser to parse the text.

2.3.6 Examples

The list of examples is also parser specific since it demonstrates phenomena that can or cannot be described by a parser implementation.

The list of examples could look like:

```
<?xml version="1.0"?>
<!DOCTYPE examples SYSTEM "examples.dtd">
<examples>
  <phenomenon id="phrase-vp-s-dec"
             description="root :: phrase :: vp :: satztypen :: deklarativsatz">
    <example id="1" status="success" comment="example comment"
            description="Deklarativsatz mit Komplement im Vorfeld."
            phrase="er gab ihr das buch" />
  </phenomenon>
  <phenomenon id="funct-subord"
             description="root :: funktionsw{\"o}rter :: subordinatoren">
    <example id="622" status="success" comment=""
            description="Subordinierende Konjunktion mit einem finiten Satz"
            phrase="er sang w{\"a}hrend sie schlief" />
    <example id="624" status="success"  comment=""
            description="Subordinierende Konjunktionen k{\"o}nnen extraponiert werden."
            phrase="er hat gesungen ohne sich zu beeilen" />
  </phenomenon>
  ....
</examples>
```

Chapter 3

Design Space Exploration

After discussing the task on a conceptual level in chapter 2, the technical solution can be discussed. In this chapter we will explore several implementation alternatives.

In section 3.1 alternative platforms, architectures and application designs are compared. In section 3.2 the technical implementation of the parser interface is discussed and in section 3.3 graphicaö user interface (GUI) technologies are evaluated.

3.1 Platform, Architecture and Design

In this section different architectures are compared. After defining the requirements and side constraints, candidate architectures are introduced, compared and refined into application designs. This section is summarized by explaining our choice.

3.1.1 Requirements

From section 2.2 we have a list of user interface components and processes available that need to be backed by the web application. In section 4.6 we define some constraints for the parser interface.

Achitectural Constraints

While most aspects of the system are up for discussion, there are certain characteristics of the architecture that are invariable.

- We deal with a web interface. The client is a web browser.
- We realize the project as a client server application. Generally, natural language parsers depend on vast amounts of resources (lexicon data, grammar) and it is not feasible to load all data statically to the client. There must be at least a a minimal, dynamic server component that reduces the data flood before transmitting it to the client.
- The client server architecture is reinforced by the requirement that the parser should be pluggable, allowing for parsers written in any programming language. Without a server component, the parser would need to run on the client. This would impose unrealistic requirements on the available client technology.

- The parser can be written in any programming language and has a vastly different application profile than the web application. For these reasons the web server application must be separated from the parser.

System Components

As a consequence of the invariables discussed above the system consists of at least three components.

Web Client: The web browser is a fix component in our architecture. It allows only for two degrees of freedom: The choice of web technologies and the choice of the amount of business logic that should run on the client side rather than on the server side. Still it offers the choice between asynchronous requests or classic synchronous HTTP requests.

On an architectural level, these degrees of freedom are not very interesting and we have to accept the clients limitations as constraints. The spectrum of web browsers is very wide and very old versions lacking modern features are still in use. Luckily, the vast majority of users is using modern web browsers and we will target for A-grade browsers [4] for our web interface.

The choice of web technologies will be discussed in section 3.3.

Web Server Application: The application running on the web server is the key component of the web interface for natural language parses. It is handling the parse requests, serving the content to be rendered by the client and, depending on the chosen architecture, it can handle most concerns of the system.

Parser: The parser component is independent of the web interface. It consists of an existing parser which is adapted to run in a process that can accept parse requests by the web server application.

For this discussion, it is sufficient to say that the parser is a separate component that implements any required protocol to handle parse and lexcon check requests.

Required Services and Components

These are the requirements for the complete parse system on an architectural level and werefine then by adding design level requirements as well:

- The web server application is serving dynamic and static user interface resources such that they can be rendered and processed by the client.
- The client or the server need to provide services that can be consumed by the user interface. This includes thesaurus lookups, auto completion requests and sorting of parse results.
- The server needs to handle parse and lexicon check request by delegating them to the parser.
- The server and the parser need to implement some protocol such that they can exchange parse requests and results.

- The web application needs some form of persistent cache to store parse results between HTTP requests. The cache can either be on the client side or on the server side.
- If the parser provides a lexicon check service it needs average response times for such requests shall stay beylow a second or two. Otherwize the service is worthless and one should restrict the parser to handling only normal parse requests.
- The Web server application should be easy to deploy and have low hardware and software requirements.

The list of requirements can be completed by adding general application design rules that follow common application design patterns.

- The web server application should provide a logging service to log exceptions and undesired events.
- A candidate framework should include means to process XML documents efficiently.
- It should include some kind of persistent storage.
- It must provide UNICODE-safe string operations.
- A model view controller (MVC) design for the HTTP request handling would be desirable.
- Additional to synchronous HTTP request handling, the framework should also provide means to handle asynchronous requests.
- Further infrastructure should include URL routing and input validation filters.
- A candidate framework should be easy to learn, well organized and well documented.

3.1.2 Comparison

It makes sense to combine the discussion of architectures with a discussion of the platforms that those architectures are based upon since the platform provides a frame for the architecture and can exercise a major influence on the application design.

Platforms

Two of the most popular web development platforms have been considered for this web interface:

- “Java” in a servlet container provides a solid platform for scalable and high performance web applications. A large collection of high quality libraries and application frameworks for various needs and a mature language complete the portfolio.

  Deployment of “Java” applications is not trivial though since Java servlet applications require a server with a servlet container, or one needs complete control over a server.

- PHP 5 is an object oriented scripting language especially suited for web development. Due to its popularity it is basically preinstalled and available on most web servers. If one can live without an external database system, deployment is as easy as copying the application folder into a web accessible folder.

  Various web application frameworks are available for PHP and missing features can be quickly added through library collections or coded with the help of PHP's collection of rich native functions.

  If one does not have to write highly scalable web applications, one can work with plain PHP 5. Once the performance needs rise, PHP can be cached on various levels, compensating for its interpreted nature and for its per request nature. But both is not necessary in our case.

Persistence

A Java web application, running in a Java servlet container for instance, provides completely different characteristics than a scripting language like PHP getting invoked through a common gateway interface (CGI) by a web server. The Java application is running as a permanent process while the PHP application is interpreted and invoked for every HTTP request again.

This also opens different solutions for persistence. While a Java application can store data in process (in memory), a PHP solution depends on non-standard PHP extensions to provide a workaround, like a separate "memcached" server (a server product providing a memory-caching service).

Other than the in-process inter-request cache, most web application platforms offer similar solutions for persistence. There are file input / output, relational database management system (RDBMS) connectors and usually some adapter available for a flat file databases like SQLite.

Persistence is required for the following components:

- Parse result cache: The slowest part of the system is presumably the natural language parser. It makes sense to cache parse results between HTTP requests.

- Full forms lexicon: Backing the auto completion requests and the thesaurus service need to store the data in an optimized manner to have fast read access to achieve the low latency requirement of the auto complete feature.

- Optimized examples page performance: Since the examples data is static and parsed from an XML file, it makes sense to persistently cache examples data that is ready to be rendered.

- Optimized grammar processing performance: The grammar is used when processing and presenting parse result data. The grammar is of static nature and needs to be parsed from an XML file. It would make sense to cache the grammar data after it has been read and parsed from XML.

- In memory design: Depending on the platform and available system resources and the size of the thesaurus and the full forms lexicon, all data can be kept in memory, if an in memory cache is available.

Alternative Architectures

Based on the above mentioned platforms we can now compare alternatives for the server side system architectures.

Monolithic architecture: It consists of a single server component running the application and the parser in the same process. This can only be realized if the application and the parser are written in the same language or for the same platform (like Java, .Net). Furthermore, this architecture buys simplicity in design by merging two components that have different application profiles. A common natural language parser has considerable memory requirements ensuring that such a monolithic application could not get enough system resources on a shared web server and thus needs a dedicated server.

Two tier Java architecture: It has several advantages. Java applications tend to be highly maintainable. The separation of the parser and the application allow for proper modularity. And the costs for deployment are considerably lower than with the monolithic architecture since the servlet part of the system is much ligher and can run in any Java servlet engine enabled server.

Two tier PHP architecture with file persistence: It is a very maintainable and agile solution. Thanks to PHP's popularity and using a highly portable file based persistence, this is a very portable architecture that can easily be deployed.

Two tier PHP architecture with RDBMS persistence: It offers improved performance compared to the file persistence based solution. For our needs this performance improvement would barely be noticeable. And it comes at the price of increased deployment and maintenance costs. PHP / MySQL combinations are more widely available than Java servlet containers though.

Three tier Java/PHP architecture based on Heart of Gold (HoG) It benefits from the reuse of an existing domain related framework. An adapted version of HoG would serve as a middle ware between a front end application and the parser. All services would be provided by HoG and the front end application could be a lightweight PHP application. On the negative side there would be three different components to maintain with a PHP/Java hybrid environment. Alternatively the front end could also be a lightweight Java servlet but at the cost of a more complex deployment.

Architecture	Monolithic	Two Tier A	Two Tier B	Two Tier C	Three Tier
Platform	Java	Java	PHP	PHP	Java / HoG
Persistence	In Process	In Process	File Cache	External RDBMS	In Process
Maintenance	++	++	+	+	-
Deployment	–	-	++	+	-
Modularity	–	++	++	++	++

Table 3.1: Comparison matrix for system architectures / platform combinations

Conclusion: Two different architectures clearly stand out. On the one hand it is the two tier PHP architecture with file based persistence. It has no real disadvantages that we would be concerned about. On the other hand there is the two tier Java architecture that only got negative points because Java applications are more difficult to deploy in a shared web hosting environment.

Comparing Frameworks

Several web application frameworks have been evaluated to find one that meets our requirements and task best. All of them provide at least basic components to build our business logic and services upon. But there are also other qualities to be considered.

Java Web Application Frameworks: From the Java world, WebWork, Wicket, Stripes, Tapestry, RIFE, Seam and combinations with the Spring framework have been considered.

While most of them offer excellent code quality and well designed frameworks, not all of them have sufficient documentation to be considered seriously and compared to some of the better PHP web application frameworks. Most Java frameworks have a too steep learning curve with many idiosyncrasies and a high level of abstraction.

For our task, designing and implementing a small application in a short time and then handing of the resulting application to a new maintainer, it is imperative that the framework is well documented and easily accessible to new maintainers without investing days for learning.

PHP Web Application Frameworks: A wide spectrum of PHP web frameworks is available. We are mainly interested in general purpose frameworks. Interesting solutions in this area are CakePHP, CodeIgniter, Prado, Symphony and the Zend Framework.

Given that we are interested in a light, modular, well documented, well designed framework that rather provides functionality than setting limits, Zend Framework is the clear winner.

Full stack frameworks like the heavy Symphony or the light but ancient CakePHP (PHP 4) and the rather modern Prado have been ruled out because of being too massive by their nature.

CodeIgniter on the other hand left an underwhelmed impression due its lack of proper unicode support.

Zend Framework is a modern PHP 5 framework with excellent documentation and design. Each component of the framework can be used independentely and there are various ways to adapt a specific component to our needs.

Client Server Models

After finding a system architecture and choosing a web application framework for the server component, it is time to evaluate the client server models.

In section 3.1.1 we have established some basic constraints for the client server interaction. The user interface is rendered by the client and the server handles parse requests requested by the client.

But we have yet to define how thick the client should be, i.e. how much of the application code should run on the client instead of on the server.

The amount of data to provide a thesaurus and auto completion service is generally too large to be transfered to the client completely. It makes sense to keep this data on the server and have the server provide those services.

Keeping most of the code in a single language and keeping it well organized in a single place on the server provides several advantages over mixing server side scripting too much with client side scripting.

As a general guideline, we will keep the business logic and most data on the server and only make exceptions where the user experience benefits substantially from asynchronous requests, low latency operations and no page reloads.

3.1.3 Summary

In this section we have derived constraints for the system architecture and refined the design by comparing different platforms, architectures and, finally, application frameworks. A two tier server side architecture with the Zend Framework, a PHP 5 application framework, forming the core of the planned web application server. The application server will be responsible to handle any parse requests, to provide services as the thesaurus and auto completion abd to serve the user interface to the client. The parser is running as a separate server.

3.2 Parser Application Interface

XML was chosen early on in the design process as the message format of our choice (see section 4.6). Since there seem to be no established standards we defined our own XML schema definition to represent a set of parse results including feature structure data (see section 4.6).

The protocol chosen to communicate between the web server application and the parser server is *XML-RPC*. It is strikingly simple and yet powerful enough and very well supported. XML-RPC client and server libraries are available for most programming languages. For instance, the Zend Framework includes client and server components for XML-RPC allowing for very quick prototyping of the interface.

3.3 GUI Technologies

In section **??** the user interface design has been discussed on a conceptual level and all user interface components have been introduced.

In this section we will look at the technical implementation, answering the question how these user interfaces can be implemented using common web technologies.

But first we will discuss the available web technologies.

3.3.1 Client Side Web Technoligies

Some web technoligies are standardized, some are proprietary. Some standards get hardly adopted and some proprietary technologies have great market adoption.

Our goal is to provide a good user experience for A-grade browsers [4], if possible without the use of any proprietary solutions.

Here is a short overview of the available technologies:

- *HTML/XHTML* (Hypertext Markup Language) is the basic building block of all web interfaces. It is available on all clients. HTML is very flexible, but it falls short when it comes to animations, interactivity or vector graphics.

- *CSS* (Cascading Style Sheets) is a simple mechanism to add style to web documents. It allows for separating form from content.

- *Images* can be displayed in any web document. Image maps allow for adding several hyperlinks in different areas of the image.

- *JavaScript* is the most wildly adopted client side scripting language. It allows for adding behavior and interactivity to the client.

- *DOM* (Document Object Model) is a language independent form to represent HTML and XML documents. DOM manipulation is often used by JavaScript to achieve certain effects.

- *XHR* (XmlHttpRequest) is a special JavaScript object that allows to make asynchronous HTTP requests in the background when a page has already been loaded. This allows for instance for introducing interactive user elements where data is loaded from the server on demand.

- *SVG* (Scalable Vector Graphics) is an XML specification for describing two dimensional vector graphics. SVG is natively supported in more and more modern browsers with the prominent exception of Microsoft Internet Explorer.

- *VML* (Vector Markup Language) is an older XML based vector graphics format and mainly supported by Microsoft. Microsoft Internet Explorer supports VML.

- *Canvas* is a HTML 5 element (most browsers today support HTML 4) which adds a drawable region to the document. It has just recently be adopted by some major browsers and it is still not supported in any version of Microsoft Internet Explorer.

- *Adobe Flash* a wildly adopted browser plugin to add multimedia and scripting capabilities to the browser. Flash's runtime supports vector graphics and animations. Although the runtime is available for free, it is a proprietery product and authoring tools are not free of costs.

- *Microsoft Silverlight* is a new browser plugin similar to Adobe Flash.

On the Importance of Libraries

It is important to note that web user interface design can be a daunting task when doing everything from scratch without the help of well tested libraries and frameworks. Fighting with cross-browser incompatibilities can quickly absorb the bulk of the invested time and progress can be unpredictable.

Thus one should use well tested and highly compatible JavaScript and CSS frameworks and stick to standards compatible XHTML markup to avoid any suprises.

3.3.2 Layout

For the layout of the pages a site wide template is offered and the CSS framework “blueprint” is used to quickly layout all elements on a grid. A task that would be non-trivial without the use of a CSS framework.

3.3.3 Input Process

The *input form* and the *thesaurus* form can both be backed by regular “YUI” (Yahoo! User Interface Library) “AutoComplete” widgets. YUI is an open-source JavaScript / CSS framework. The “AutoComplete” widget can listen for input events on the HTML input form, requests auto completion results from a specified service via XHR and displays the auto completion results via DOM manipulation below the input prompt.

The server side services return the results in a newline separated format as UTF-8 encoded plain text.

Input Filtering

To prevent the input of any illegal character into the input field, a JavaScript event handler can listen for input events, check the input text and update the input form through DOM manipulation with a filtered version of the text if necessary.

The filter rules are specified as a regular expression that can be configured in the web application.

Lexicon Checks

For the lexicon checks another input event listener is registered on the input form. The event handler uses XHR to request lexicon check results from the server and it displays the check results via DOM manipulation above the input form.

Similarely, all other features in the input process can be handled with XHR, DOM manipulation and DOM event listeners.

3.3.4 Parse Browser

The parse browser (see figure 2.10) features a couple of technical challenges. First there is the disambiguation widget (see figure 2.11) with its hover effects and then there are the sorting menus.

Disambiguation Widget

From early mockups it was clear that it should be possible to render the disambiguation widget with XHTML and CSS. An approach with nested XHTML “¡span¿...¡/span¿” elements and supporting CSS classes has been undertaken with various attempts to get a solution that works well in all major browsers but the attempt ultimately failed. Although semantically less correct, replacing the nested “¡span¿” elements with nested XHTML tables might turn out to be a very simple, robust and well-supported solution.

Adding color coded parts of speech tagging could not be easier. Thanks to CSS it is sufficient to assign each word element in the disambiguation widget the right CSS class adding the required font color.

Hover Effect

The hover effect illustrated in figure 2.11 can be added using JavaScript DOM event listeners and DOM manipulation.

Using CSS only: This solution cannot work. Although CSS features the dynamic pseudo class ":hover" when an element is being hovered by the mouse pointer, it does not provide any ancestor selectors that would be needed to provide any meaningful hover effect. Since the disambiguation widget is a structure of nested tables, a nested table for each phrase, the mouse pointer hovers not only a specific phrase, according to CSS rules it hovers all container elements as well. Thus if one would assign a different color for hovered phrase ("¡table¿") elements, half of the disambiguation widget would change the color since one hovers several outer phrases when hovering an inner phrase. With a CSS ancestor selector one could specify that only the inner most hovered element should be highlighted. But such a selector is neither available in CSS version 2 nor in version 3.

Using JavaScript: This solution in combination with the DOM events "onMouseOver" and "onMouseOut" for this effect failed in an interesting way. The JavaScript library "jQuery" offers the function ".hover(doSomethingOnMouseOver, doSomethingElseOnMouseOut)" which seems to be the perfect tool to implement the effect. Since the related event onMouseOut does not provide any information about the element that is entered when leaving another element, we have to maintain a stack of elements that we enter and leave. On-mouse-over we push the element that we enter to the stack and add the hover effect to the element that we just entered. When leaving an element, we pop an element from the stack and add the hover effect to that element.

This works perfectly in theory and it works pretty fine in most cases on a fast client machine as well. But when moving very fast over many phrase edges, the hover effect might show erratic results. It might add the hover effect to the wrong phrase or have massive delays in following the mouse pointer. On slow client machines the effect stops working correctly even with slower mouse movements.

This approach failed because the time resolution of the involved DOM events would not be high enough. The approach is based on the assumption that the event listeners would get called reliably everytime an element was entered or left. That is not the case.

Conclusion: The final solution is again based on JavaScript. But instead of relying on a perfect DOM that should notify us for all DOM events and instead of requiring high performing client machines, we do away with the element stack and listen for other events.

The "onMouseMove" event should be used with caution since it gets fired on every mouse movement, even if it moves just a single pixel. But the event has a major advantage over "onMouseOver" and "onMouseOut": It includes a reference to the element that the mouse is currently hovering. This is all the information we need. To limit the performance degradation, we listen only for mouse movement events when the mouse is located over a disambiguation widget.

Sorting Interface

The sorting interface consists of a panel with instructions and a list of all active sorting criteria and options to remove criteria or change the order in which sorting criteria get applied.

This is also realized in JavaScript and the challange is to keep track of sorting criterion specific code in as few places as possible.

The sorting interface would not be complete without the interactive sorting menu that are shown when opening the context menu over a word in a disambiguation widget. Luckily, YUI provides an efficient way to register a context menu callback function for many objects at once. Note that large numbers of event listeners could quickly become a performance bottleneck.

3.3.5 Derivation Tree

The derivation tree is considered to be the biggest challenge. First, one needs to process the results with a layouting algorithm to determine the position of each tree node and the connecting lines.

Then there is the problem of rendering diagonal lines in a browser. XHTML only allows for rectangular lines. And the problem of adding interactivity to the tree nodes.

Alternative Rendering Methods

A lot of alternatives have been considered.

- Using basic XHTML doing without any pretty diagonal lines.
- Using SVG and thus forcing the majority of users to switch browsers (or rather not use our application).
- Going with Flash, a proprietary runtime library (vendor lock-in etc).
- Using CSS / JavaScript tricks to generate diagonal lines.
- Using server side generated images delegating the rendering to established tools like “graphviz”. Using image maps to add basic interactivity.
- Using Dojox.gfx might be the best solution. It is a JavaScript based vector graphics library which abstracts the implementations (SVG, VML, Silverlight) from the programming interface and auto detects the right renderer.

Dojox.gfx offers another advantage over plain SVG: It provides connectors to add JavaScript event listeners to SVG/VML elements.

Tree Layouting

Suprisingly, tree layouting libraries or algorithm implementations in JavaScript are a rare breed. Luckily, there is a freely available JavaScript based tree widget wich implements “a node positioning algorithm for general trees” [5].

The rendering code of the widget was replaced with dojox.gfx and interactivity has been added by adding event listeners and showing superimposed windows with YUI.

Chapter 4

Implementation

We introduced the underlying interaction processes in chapter 2 and determined the application design and platform has been in chapter 3. In this chapter we now discuss the actual implementation of the system.

Section 4.1 introduces the system architecture and gives an overview of the subcomponents of the system. Section 4.2 highlights the core components of the web application that are part of the lifecycle of a parse process. Section 4.3 discusses all other components of the web application before the components of the user interface are documented in section 4.4 and its processes in section 4.5. The parser interface and parser specific elements are discussed in section 4.6, the chapter is closed by covering the deployment in section 4.7 and debugging procedures in section 4.8.

4.1 System Architecture

Figure 4.1 gives an overview of the whole system.

Overall it is a three tier architecture consisting of three main components:

- Client (Web Browser): Renders the user interface provides interactivity through data-driven widgets and asynchronous requests.
- Web Server Application: Lightweight request handler and controlling (based on the two tier architecture; see chapter 3) application logic.
- Parser Server: Independent, remote process handling parse requests.

4.1.1 Web Server Application

The web application consists of the application running on the server as well as of the client. The client renders the user interface and part of the application logic actually runs in the browser in the form of JavaScript.

Figure 4.1 shows the *layered architecture* of the application strictly separating presentation from the domain model and the parse service layer. Each layer can only interact with the layer directly above and below it. Additionally, an object from a specific layer is not allowed to start any interaction with the layer above it. It is only allowed to get called by the upper layer.

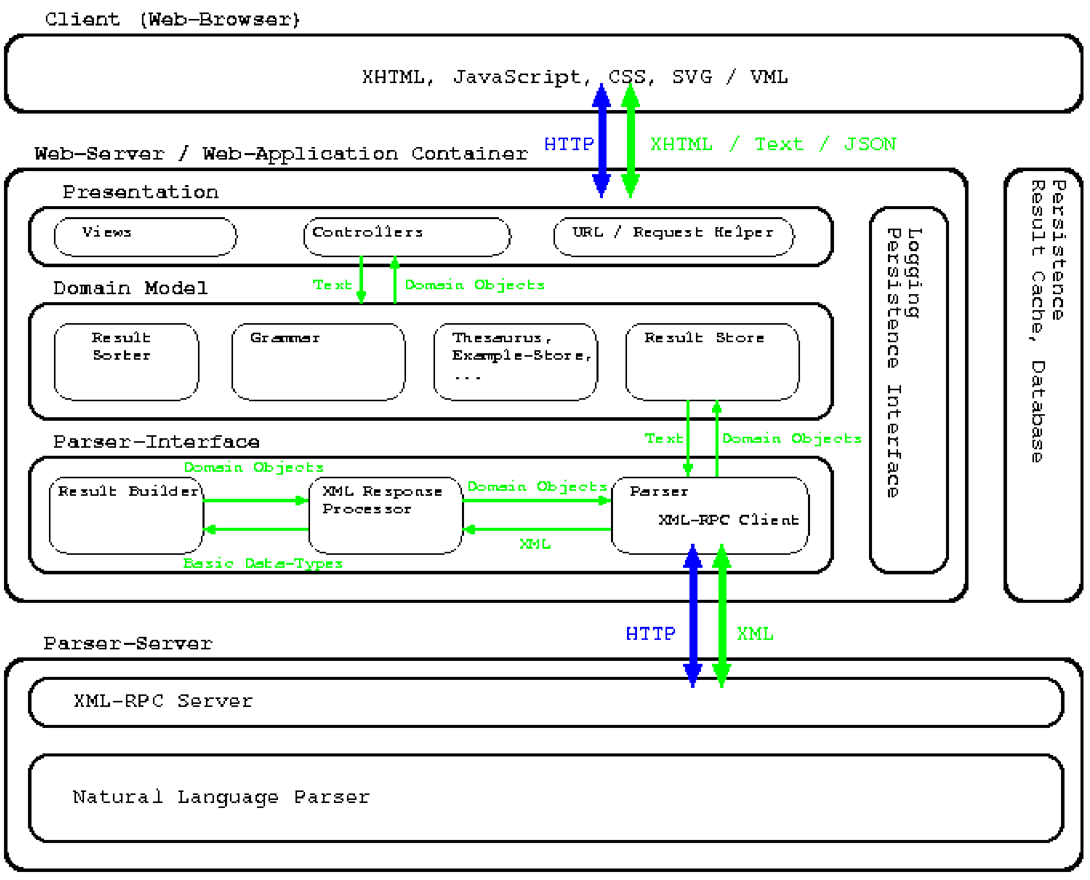


Figure 4.1: Overview of the system design

There are also some cross cutting aspects like logging, exception handling and the persistent cache.

Overview

Before discussing the application in detail, this section will go quickly through each layer and component.

Client Side Presentation Layer: The client renders the XHTML pages that are sent to it by the server component of the application. The XHTML can include CSS, JavaScript; images and the JavaScript can in turn generate SVG or VML.

Server Side Presentation Layer: The frontend of the web application uses the model view controller pattern (see section 4.1.1). A controller handles the process by loading data and preparing it to be rendered by the view. A view is usually just a XHTML template with some placeholders that can be dynamically replaced with the loaded data. There is usually a one to one correspondence between controllers and views.

The application has a default / index controller which is responsible to render the input form. Then there is a parse controller which is called by the input form, delegates the parse request to the domain model and redirects the user to the parse browser. The results controller is responsible to load and display the parse results for a given phrase.

There are also controllers for the thesaurus, the auto completion and for lexicon check services. They all have in common that they don't have a view. These are controllers that generate immediate output without any XHTML markup.

And then there are the controllers and views for the "examples", "about", "error" and for the "help" page.

Domain Model: The domain model is home to the core component of the application, the result store. The result store is what the controllers call when they want to parse a phrase or load the parse results. And the result store is the connection between the presentation and the parsing layer. It delegates parse reqiests to the parse layer as necessary. The result store is also caching all results persistently.

Other components in the domain model are the result sorter, the grammar processor, the thesaurus, the full forms lexicon and the examples processor.

A compagnion of the thesaurus and the full forms lexicon is the persistent database which resides outside of the application but is accessed through the domain layer.

Parsing Layer: The parsing layer is concerned with parsing phrases and returning parse results to the results store.

An XML-RPC parser client sends the parse requests to the remote parser server. When the response arrives, it will be processed by the XML response processor and assembled to a parse result object by the parse result builder.

Data Structures: There are very few data structures involved in the application. As illustrated in figure 4.1, requests come in the form of UTF-8 encoded phrases. The controller requests parse results for the given phrase. The domain model returns lists of "ParseResult" instances. ParseResult is used when ever possible. If the result store does not have a result yet, it forwards the phrase to the parser and the parser will return a ParseResult. The XML schema of the parser interface will be discussed in section 4.6

Concepts

Before discussing the components in detail, a few fundamental and some cross-cutting concepts and patterns need to be introduced.

Model View Controller Pattern: Figure 4.2 illustrates a simplified data flow diagram of a typical request in the model view controller (MVC) architecture that is used in the application. The MVC component provided by the Zend Framework installs a front controller that controls the whole requesting handling process. A URL router maps the requested URLs to the corresponding controller and action name and the dispatcher calls the specified action on the given controller.

The controller receives control over the request at that point and starts processing the request. The parse browser controller for instance might parse the requested result identifier

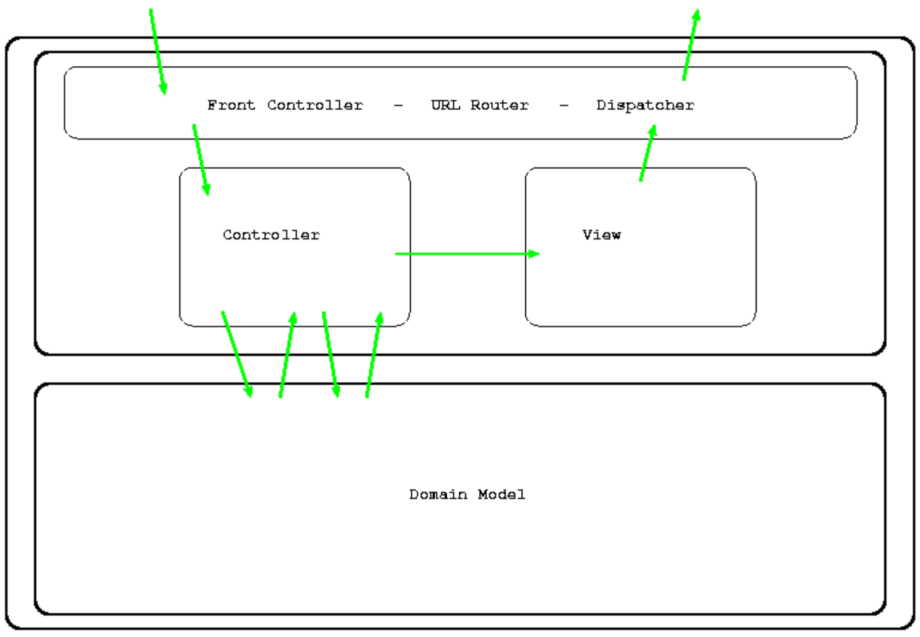


Figure 4.2: Simplified diagram of the model view controller pattern

from the request parameters and then attempt to load the corresponding parse result from the domain model.

Once the controller has loaded all required data and prepared the data for the view, the control is given to the view which uses the prepared data to produce a dynamic XHTML response and the resulting XHTML document is printed to the output stream and transported to the client via HTTP over TCP/IP.

The clear separation of concerns gives the MVC pattern its name.

Security: The application does not deal with sensitive data, but a secure design is still imperative. If not properly secured, the application could be a way to intrude the internal network through its XML-RPC parser client. Similarely, the file system access and database queries have to be secured to avoid any bad surprises.

- All input data is interpreted as UTF-8 and all output data is declared as UTF-8 leaving no room for speculation.
- All input is filtered as strictly as possible. Yet, it is up to the administrator of the application to limit the list of legal characters reasonably.
- Prepared database statements are used where possible such that the data is bound and SQL injections made impossible. In other places, the data is escaped properly before using it as parameter.
- Cross site scripting is prevented by escaping all strings that are prepared for display.

- No errors or stack traces are shown to the end user unless the application is in debug mode.
- All unexpected events get logged persistently.
- By default, the parser server address cannot be specified by the end user. If the cookie feature is active though, the white-list filter must be kept strict enough to avoid any problems.

Full UNICODE Support: The application treats all strings as UTF-8 encoded text, be it in JavaScript or at the XML-RPC interface.

Configuration: A central configuration file allows for configuring the most important properties.

4.1.2 Parser Server

As illustrated in figure 4.1, the parser server consists of three components.

- A *HTTP/XML-RPC Server* to get notified of parse requests and to have a simple way to return the results.
- An adapter which translates between the parser's original result format and the schema expected by our application.
- An existing parser.

4.2 Parse Model

The parse model consists of all classes that are concerned with "ParseResult" objects.

4.2.1 Parse Result

"ParseResult" (parse result) is a value object representing a specific reading of the parsed text (see chapter 2). It is used on the server side of the presentation layer to extract data that is required for all the different visualizations.

A parse allows for object oriented access to all data returned by the parser. Additionally to methods to retrieve tree nodes, each parse result has a list of phrases and a list of words.

A parse result word is defined as the node with the first lexical rule in the sequence of rule applications.

A parse result phrase is derived by combining the grammar with the parse result data. Each phrasal node implies a new phrase unless the node itself is a head.

4.2.2 Result Store

For the presentation layer, the result store is the source for all parse results.

The result store is a transparent cache for parse results. The result store is a client of the persistent cache service of the application. It stores and loads parse results from the cache.

If there is a cache miss, the result store loads a parser instance from the parser factory, waits for the results to come in, stores the results in the cache and then returns the results.

The result store not only handles normal parse requests, but also lexicon check requests. There is an option to disable persistent caching for lexicon checks since it is questionable whether caching improves performance (in this case since caching for lexicon checks already needs to be implemented on the parser server).

4.2.3 Result Sorter

The result sorter accepts a list of parse results and an ordered list of sort criteria before sorting the list of parse results accordingly.

Sort Criteria Factory

There are currently four different sort criterion implementations each concerned with another property of a parse result.

The sort criteria factory [6] translates request query parameters to sort criterion instances ready to be used by the result sorter.

4.2.4 Parser

The parser layer of the application gets called by the result store for normal parse and for lexicon check requests.

The parser interface is defined separately from the XML-RPC parser client implementation. The common way to get a parser instance is to use the parser factory.

XML-RPC Parser Client

The XML-RPC parser client is delegating all requests to an instance of a Zend XML-RPC client. Furthermore, it overrides the default HTTP client to define different timeouts for the initial attempt to establish a socket connection and for the actual HTTP transaction.

It does not make sense to let the user wait 60 seconds in front of the browser if the parser server is down in the first place. On the other hand, it might take up to 60 seconds until the parser replies to the parse request with the results.

The XML-RPC parser client translates all HTTP and XML-RPC related exceptions from the Zend framework to application specific exceptions.

XML Parse result data is first processed and then returned as domain objects to the result store.

XML Response Processor

The XML-RPC parser client has a XML response processor to handle parse replies. The response processor validates the XML documents and parses the XML data into primitive data types. It then uses the result builder to create parse result nodes and parse results from the primitive data types.

Result Builder

The result builder is intimately familiar with parse results, parse result nodes, parse result words and parse result phrases. It creates new instances from primitive data types and it derives the list of words and phrases before it creates a new parse result.

4.3 AuxiliaryComponents

The web interface would work with just the parse model and the parse layer. It is the core of the application. But it would perform poorly without a persistent cache the user interface would not be the same without the auto completion and thesuaurs services.

In this section we discuss these auxiliary components.

4.3.1 Persistent Cache

The persistent cache provides a service for the whole application to store any data, be it strings or serializable objects between HTTP requests. Zend's cache component makes it possible.

The cache has no explicit size limit, but each cache entry has a configurable lifetime.

The result store periodically triggers a clean process of the cache when a new cache entry is stored. The probability of a cache clean operation is chosen such that the cache size should not increase substantially over time.

4.3.2 Logging

Logging allows for taking notice of unexpected events and for acting upon it. The application is using the logging component of the Zend Framework with its BSD syslog-style log levels. The log level can be configured through the configuration file.

4.3.3 Exception Handling

Common exceptions from third party components like the Zend Framework are caught and rethrown as application specific exceptions.

All exceptions are caught in the end, logged and a user friendly error page is shown. For some likely error events like parse errors or when the parser goes offline, the error page shows specific error messages and suggestions.

4.3.4 Database

An SQLite database is available to all classes in the domain model.

4.3.5 Thesaurus

We use a very simple thesaurus implementation, only offering a plain list of words matching a search term. A simple two column database table with an index (primary key) on the term and having the matching words as space separated list in the second column has been chosen to yield the best query performance for our thesaurus reuests.

Terms are all stored in lower case because as of this writing, SQLite would not handle UTF-8 correctly. SQLite compares strings case-sensitively and its UPPER / LOWER functions only work correctly in the ASCII range.

Therefore, we store all thesaurus data in lower case and convert the search term with a UTF-8 safe method to lower case in PHP before querying for results.

4.3.6 Full Forms Lexicon

We have two different search queries on the full forms lexicon. Thesaurus requests need an exact match to see if a requested term is in the list of known words. Auto completion requests need all lexicon entries that start with a given prefix. This is an interesting requirement since an indexed SQLite column searched with a (LIKE 'prefix%') clause looks like the perfect and simple solution to this problem.

For the database schema, a simple 1 column table with a primary key was chosen. Each row consists of a full form.

As with the thesaurus, data needs to be stored in lower case to work around SQLite's shortcomings concerning UTF-8.

4.3.7 Grammar Processor

The grammar processor is concerned with the parsing of the grammar.xml file into a native data stucture and with providing cached access to this grammar data to any consumer. The grammar.xml file is validated against the XML schema when parsed a new.

4.3.8 Examples Processor

Similarely to the grammar processor, the examples processor is concerned with the management of examples data. Examples are read from an examples.xml file, validated against the examples.dtd document type definition and cached for its consumers.

4.3.9 Application

The application instance is the single global variable in the web application. It provides getters to common services like the cache, the URL generator, the logger and the database. Thereby it is also offering a natural way to defer the loading of particular subsystems as long as possible.

4.4 User Interface Components

The visible components of the user interface design have been discussed in section **??** and the technical aspects involved in implementing these components have been discussed in section 3.3.

Before taking a detailed look at the processes involved in the user interaction, let us examine some of the hidden, server side components and principles involved in the handling of HTTP request. Client side components are discussed last in section 4.4.6.

4.4.1 Controllers

As explained in section 4.1.1 the controllers are responsible to process the HTTP requests. Each controller in the Zend framework can have multiple *actions* and has a default action called “index” which gets invoked if no action is specified in the request URL. And if the name of the controller is not specified either, the framework invoked the default action on the default controller.

Our application is configured to invoke the “index” controller by default. It is responsible to display the input form.

Base Conroller vs. Immediate Controller

There are two types of controllers in our web application. The classic one works as illustrated in figure 4.2. But to serve XHR responses, the XHTML view has to be disabled and data has to be returned in a compact and parsable format like newline and tab delimited lists or JavaScript Object Notation (JSON) can be used.

To support these two common use cases, all classic controllers can extend the abstract base controller (BaseController) and XHR handling controllers can extend the immediate controller (ImmediateController).

Base Controller The base controller loads common data that is used by all views. The data used to generate the navigation menu, the page title and common view variables get assigned in the base controller. The context sensitive help page URL is determined here as well.

Immediate Controller The immediate controller configures the MVC framework to omit the view rendering stage of the request handling. And it sets common HTTP headers to properly declare the response as unicode encoded text as opposed to XHTML that is generally the content type of all other responses.

4.4.2 Views

A view is a text file with a “.phtml” file extension. The extension is indicative of what can content can be used in the file. One can treat a view file as a normal XHTML file and augment it with PHP code for dynamic parts.

Typically, one uses XHTML throughout the view file and adds PHP segments to include values for placeholders or to add basic conditional behavior or to output contents in a loop.

In our application there is mostly a one to one relationship between controllers and views. With the exception of the immediate controllers, each controller does not specify a special view name and lets the framework render the view with the same name as the controller by default.

The views folder of the application is organized as follows:

```
views/
  scripts/
    about/
      index.phtml
    ...
```

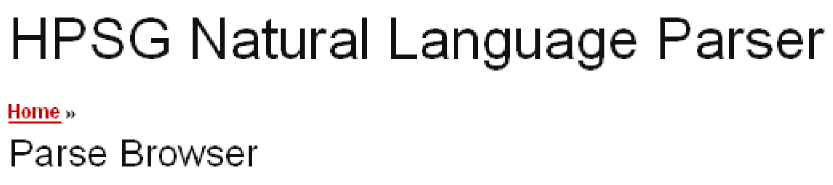
```
help/
   index.phtml
   resultbrowser.phtml
   resultviewer.phtml
index/
  index.phtml
...
```

The framework looks for a folder with the same name as the controller and looks in there for a view template file that has the same name as the controller action.

Common Look

All views served by the application share a common look (layout and style) by including a basic set of style sheets a common header and a common footer.

To give each view freedom over its XHTML "<head>" element content while preserving a common a look, three templates have been created that should be included in every view.


HPSG Natural Language Parser

Home »

Parse Browser

Figure 4.3: Page header.

- "headHtml" sets the page title and includes a basic set of CSS and JavaScript files. It needs to be included in the "<head>" section of every view. An example is shown in figure 4.3.
- "header" gives each view a basic layout, displays a heading, a navigation section and system links to the help, examples and to the about page. And it defines the start for the content area of each page. An example is shown in figure 4.4.
- "footer" adds a common footer text to all views and defines the end of the content area of each page.

Powered by MrParser

Figure 4.4: Page footer.

Each controller can override the values defined by the base controller for the page title, heading, etc. before the view gets rendered.

Layout

All page elements are positioned on a grid of 24 columns of 30 pixels with a 10 pixel margin between columns. each. The content grid is 950 pixels wide and horizontally centered.

An element can be positioned in the grid with CSS classes. The class “column” must be specified to declare that the grid logic should be applied to the element. The class “first” specifies that the element is the start of a new row in the grid. Similarely, “last” marks the end of a row. The width of an element can be specified with “span-x” where must be an integer between 1 and 24, and specifies the number of columns.

A row can also consist of a single element. Example:

```
<div class="column span-24">Here goes the content</div>
```

The complete CSS grid positioning system is based on an unmodified version of the “blueprint” CSS framework.

Anatomy of a View

A minimal view template is described below:

```
<?php echo $this->docType; ?>

<html <?php echo $this->htmlTagAttributes?>>
<head>
<?php require_once($this->commonHeadHtmlTemplate); ?>
</head>
<body class=" yui-skin-sam">
<?php require_once($this->headerTemplate); ?>

<p>Some content...</p>

<?php require_once($this->footerTemplate); ?>

</body>
</html>
```

Obviously one could choose different approaches to add a common header and footer that would require less code in each view. This approach has been chosen since it gives each view complete control over its contents and look, it is easy to opt out from the themed look. And all XHTML components of a page are still present in each view to keep it simple and maintainable.

4.4.3 URL Generator

The URL generator offers several convenience methods to get parameters from the request URL (or generally from the request data) and to generate all necessary URLs.

The URL generator makes the handling of URLs much easier and more efficient and secure. All parameters fetched from it are properly filtered and all URL parameters get properly encoded when generating URLs.

4.4.4 URL Routing

We are using the default configuration of the URL router that is part of the Zend controller component which means that it tries to use the popular Apache mod_rewrite server module to offer concise URLs. When mod_rewrite is not available, it falls back to PATH_INFO CGI variable. That is, with mod_rewrite one gets URLs like "/parser/results/show/resultId/15" while one gets URLs like "/parser/index.php/results/show/resultId/15" with PATH_INFO as fallback.

And both URLs are just considered to be nicer than the standard URL format "/parser/index.php?controller=results&action=show&resultId=15".

The URL router inspects the URL and informs the framework what controller and action should be invokved.

4.4.5 Redirecting on Success

A general rule in web application design is to redirect the client to another URL on a successful HTTP POST action. The idea is that a user could accidentally trigger the same action again just by refreshing the page or trying to load a page again. By redirecting parse requests to the results page on success, this requirement is met.

Another effect of redirecting parse requests is that all resources are always accessed through the same URL. The input form is shown as the default view when accessing the application, there is a URL to post parse requests to and to view the results there is another URL. If the parse controller would not redirect to the results page after handling the parse request, then the results view would be accessed through the parse URL.

In case of an error, the resulting page may show an error message and include a HTTP error code as status.

4.4.6 Client Side User Interface Components

While most client side elements are built from basic XHTML, there are also more complex elements which are discussed in this section.

Auto Complete Elements

The input page employs "YUI"'s AutoComplete widget for the auto completion feature of the input form element and for the thesaurus form element.

HTML entities and the AutoComplete widget: The default behavior of the widget is changed to fix its handling of HTML entities. By default, HTML entities are rendered correctly when displaying the auto completion results. But when selecting one of the auto completions, the value is copied to the input box as HTML code, not as the rendered text (e.g. "&amp;" instead of "&") which is counter intuitive. This is fixed by overriding the widget's "itemSelectEvent" handler.

Showing a message when there is no result for the thesaurus request is achieved by overriding the AutoComplete widget's "getResultEvent" which is fired before the XHR results are handed over to the widgets result renderer. This way the thesaurus the message "Unknown term ..." if the term that has been checked for is not in the thesaurus.

Lexicon Check Handler

The lexicon check are either triggered directly by an input event, or they are scheduled for some time in the future. They are scheduled if it detects that a lexicon check request has already been sent and that it is still waiting for the response. Or if the minimum period between requests has not been exceeded yet.

Lexicon checks are only performed for all but the last word since the last word is probably not finished yet and we do not want to trigger a lot of spurious lexicon check requests. For this to work, the text to be checked needs to be tokenized in JavaScript with a configurable list of tokenizer characters.

The actual request is performed with "YUI"'s XHR component in an asynchronous way. The handling of the response is described in section 4.5.1.

Unknown Word Handler

When the user clicks on the input page on word that is marked as unknown, a handler for unknown word gets triggered. It loads a list of suggestions for the given term via an XHR thesaurus request and displays the results using "YUI's" menu widget with a callback for menu selections to remove the unknown word and replace the word with the suggesion in the input text.

Disambiguation Widget

The disambiguation widget (see Figure 2.11) has been discussed on a conceptual level in section 2.2.4 and the technology applied to implement display has been discussed in section 3.3.4.

It is assembled completely on the server side by preparing the required data in a controller and generating the nested XHTML "<table>" elements in a view. Each phrase opens a nested table with a single row element and each word is a rendered as a column in that row.

Sorting Components

The basic structure of the sorting interface in the parse browser page is generated on the server. The actual sorting criteria listed in the interface are generated on the client via JavaScript. The server includes a JSON representation of the active sorting criteria in the page data. On page load, a JavaScript method goes through the list of sorting criteria, renders each criteria and adds it to the list of active criteria in the sort interface.

On page load, JavaScript also adds the context menu handler to all words in each disambiguation widget instance. The event handler for context menu events (right mouse click event on most platforms and in most web browsers) looks up meta data for the selected word like a list of word features (numerus, genus, etc.) and uses "YUI's" menu widget to display a list of available sorting criteria. As callback for the menu selections it registers a method that adds the selected method to the list of active sorting criteria and updates the displayed list in the sorting interface accordingly.

Derivation Tree Widget

The derivation tree widget uses the parse result data to build a tree of parse nodes and uses the grammar data referenced by parse result to get human readable information for rules and rule arguments.

The “dojox.gfx” component from the “Dojo Toolkit” is used to render the tree. The rendering toolkit abstracts the actual rendering technology from the API. It renders the tree using either SVG, VML or Microsoft Silverlight, depending on the capabilities of the client. In practise, it uses SVG on most modern browsers and uses VML in Microsoft Internet Explorer since it has no native support for SVG.

The derivation tree object has a list of node objects and provides an API to add and remove nodes, to render the tree, and to collapse and expand subtrees.

Loading the tree data is done by creating tree node objects from an array of node data and assigning proper parent child relationships.

Each tree node object is identified by a unique id, has a parent node id and a container for additional attributes like the referenced grammar rule and rule argument, its textual representation, etc.

Layouting Algorithm: The tree object implements a general layouting algorithm to position the nodes in a two dimensional area [5]. The position of the tree itself is determined by calculating the bounding box of the whole tree and shifting all elements before the rendering phase is initialized. The bounding box of an object is defined as the coordinates of all four corners of the smallest rectangular that incloses the object in question.

Detecting the size of text: Since neither “dojox.gfx” nor the underlying SVG technology can specify the bounding box that text elements require, we are using fonts with a fix width per character and estimate the dimensions of text by rendering a sample text for an instant, reading its dimensions and removing the text element again. This is done in the initialization phase when a new tree widget is instantiated. Based on the measured width per character and character height, text can be rendered with fix dimensions as well.

Displayed node elements: For each node, the widget can display the name of the node’s grammar rule, the name of the node’s grammar rule argument and its textual representation. The textual representation for phrasal nodes is not shown by default since a phrase would take too much of screen real estate.

All text is displayed in an abbreviated form if the estimated width is larger than the maximum configured width per node. Lexical nodes are excempt form this rule such that their textual representation is always shown completely.

Additionally to all the text, each inner node has a small icon showing a plus or a minus sign to collapse and expand the subtree that is spanned by the node as the root of the subtree.

Handling events: Interactivity is added to the tree by combining “dojox.gfx” API to get event sources for any rendered shape with “YUI”’s event API. Mouse click event listeners are registered for all displayed rule names and for all expand and collapse icons.

The tree’s own collapse method serves as the event handler for collapse and expand events. After updating the state of the corresponding tree node object, it runs the layouting algorithm and renders the tree again.

Click events on rule names are handled by looking up the grammar rule data and using “YUI”’s container widget to display the rule information as a superimposed window. The displayed information about the rule includes the rule type (phrasal or lexical), a list of all arguments and optionally some free form XHTML that can be specified for each rule. This additinal information is usually used to reference related publications.

Links: The link (or lines) between the nodes can be drawn in different styles. The default style, a straight line, is how the node connections of derivation trees are usually drawn. The widget can draw the links also as curved lines using cubic bezier curves or as rectangular connections avoiding diagonal lines.

Heads are visualized using a different color for the line.

Data-Driven Nature: The widget is said to be data-driven since it is loaded, configured and built exclusively from data. No server interaction is required to draw or update the tree. This allows for user interactions with low delays between the trigger event and the resulting effect.

Tree Renderer

The tree renderer is a JavaScript component that mainly consists of grammar data to be consumed by the disambiguation tree widget. It also has some dynamic configuration values for the widget like the location of the icon images.

The grammar data is kept separate from the other data that is required to build the derivation tree since it is static in nature. Every instance of a derivation tree is based on a different data set, but the grammar data is always the same and does not have to be transfered to client repeatedly.

The tree renderer controller serves this grammar data. The corresponding view of the tree renderer controller renders a JavaScript file and not a XHTML document like most other views. The JavaScript served by the tree renderer controller mainly includes all static data and logic to render the tree, i.e. the grammar data in JSON format and a JavaScript function to instantiate and configure the tree widget.

4.5 User Interface Processes

Equiped with the knowledge about the underlying concepts from section 2.2 and the server side user interface components from section 4.4 we can now look at the processes describing the application.

4.5.1 Input Process

The basic components of the input form have been discussed in section **??** and in section 4.4 and the technologies used to make it work have been discussed in section 3.3. In this section here we will discuss how the input process triggered by user actions will interact with the web application.

Basic Input Process

The basic functionality of the input page is provided by the XHTML input form. After the user has entered a sentence, the user can submit it to the parser by either pressing the parse button or by pressing the return key.

The form is configured to submit the parse request to the parse controller as a HTTP request with the HTTP POST method. Once the form gets submitted, the input process ends and the parse process begins. The parse process is described in the next section.

Using the Thesaurus

The thesaurus form element is hidden by default and the user can display it by clicking on the thesaurus prompt.

Once the user enters some text into the thesaurus form, YUI's AutoComplete widget submits an XHR request via JavaScript to the thesaurus controller.

The thesaurus controller requests resuts for the given term from the thesaurus and then returns the results as a newline separated list. As an immediate controller, it does not have any view associated with it.

The YUI AutoComplete widget receives the XHR results from the thesaurus controller and renders them below the thesaurus input element.

Working with The Input Form

Each input event for the input form triggers several event listeners. The text gets filtered, auto completed and lexicon checks are performed.

Filtering is performed when a character is entered into the input field. The text within the input form gets updated immediately with a filtered version if any of the characters in the input field are not in the range of legal characters as configured by the application.

Auto completion is performed after a configurable delay after the last input event. YUI's AutoComplete widget performs a XHR request to the auto complete controller.

The auto complete controller tokenizes the sentence to get the last word and requests all terms from the full forms lexicon that start with the given prefix. As an immediate controller, it returns the results directly as a newline delimited list and no view gets rendered.

The AutoComplete widget renders the XHR results as a list of possible auto completions below the input form and the user can select one of them or just continue to type upon which the auto completions are hidden again.

A lexicon check is triggered with a lower frequency than auto completion requests since the server side processing can take much longer.

The lexicon check controller delegates the request to the result store which invokes the lexicon check method on the parser.

The parser returns a list of unknown words and the list is returned to the client by the lexicon check controller in JSON format.

The client code then displays the lexicon check results above the input form. If there are no unknown words, a short message indicates the success. In case there are unknown words, it shows them as a list of links.

When the user clicks on an unknown word link, it triggers an XHR thesaurus lookup request and renders the results as suggestions below the unknown word.

When the user selects one of the suggested corrections, the corresponding word in the input word is replaced with the selected suggestion and the unknown word gets removed from the list of unknown words.

Alternatively, the user can continue to type and the lexicon check results gets updated after the next lexicon check request has been performed.

4.5.2 Parse Process

The parse controller filters the specified sentence to ensure that it has only charactes that are allowed by the parser. It then asks the result store to parse the sentence.

The result store invokes the parser if the parse results are not cached yet. The XML-RPC parse client posts a parse request to the parser server and then returns the results to the result store where they get cached.

Finally, the parse controller redirects the user automatically to the parse browser page.

4.5.3 Disambiguation Process

The disambiguation process is invoked by requesting the results controller. The results controller then displays the parse browser. The user can then view the results and sort them. The process is finished once the user returns to the input form or continues to inspect the results in detail by browsing to the result viewer.

The results controller displays the parse browser as a default action. It loads the parse results from the results store and sorts them by parse weight by default. It then prepares data that is required to display the parse browser and the view is used to renders the page.

Once the parse browser is rendered, the user is confronted with either no results, a list of unknown words, a single reading or a list of readings.

In case of no results and unknown words, the user can return to the input form to modify the sentence.

If there is only a single reading, it is displayed using the disambiguation widget as shown in figure 2.11 and the user can follow a link to inspect the result in the result viewer.

If there are multiple readings, they are shown in a vertical list and each is displayed using a separate instance of the disambiguation widget. The user can add sorting criteria by accessing the context menu over a word in the disambiguation widget and selecting one of the available criteria.

Sorting criteria can be handled at the bottom of the page. One can remove a criterion or move it up and down to specify that it should sort first or last by this criterion and then by others. After the sorting criteria have been changed, the user can request the readings to be sorted according to the new criteria. When the user clicks on the "Sort now!" link, it submits a HTTP request to the results controller with the sorting criterions as a request parameter.

After loading the results again from the result store, the results controller uses the sort criteria factory (SortCriteriaFactory) to translate the textual representation of the sort criteria into a list of sort crition (SortCriterion) instances. It then uses the parse result sorter to sort the list of parse results according to the given list of sort criteria and renders the parse browser with the sorted list of readings.

This process continues as long as the user chooses to change the sorting criteria.

4.5.4 Inspection Process

The process of the inspection of a specific reading is centered around the derivation tree widget.

The user requests the results page for a specific reading by invoking the show action of the results controller. The controller loads the parse result from the result store and and prepares the data to be rendered.

The rendering process is divided into two parts. The actual layouting and rendering of the derivation tree happens in the web browser. The server side part consists of including the necessary JavaScript data such that the JavaScript that runs on the client can build and render the tree. All data is included in the page using the JSON format such that JavaScript can access the data directly. The server side code also includes the resource locator to the necessary grammar data in the resulting page.

The first thing the cllient does when loading the page is requesting the grammar data from the application. This is discussed in detailed in section 4.4.6. Once the grammar has been loaded, the data-driven derivation tree widget is instantiated as soon as the web browser fires the "onLoad" event indicating that all JavaScrpt resources referenced in the "<head>" section have been retrieved.

The derivation tree widget uses the parse result data to build and render a tree of parse nodes.

When the user clicks on an plus or minus icon that is displayed for all inner nodes, the subtree of the corresponding node is expanded and collapsed respectively.

When clicking on a grammar rule, the event listener displays a superimposed window with further information about the rule.

Additional tree controls allow to reset the view and to show or hide all morphological information, i.e. to expand all nodes or to display only all phrasal nodes and the first lexical node in the sequence of rule applications.

4.5.5 Additional User Interface Components

The discussion of user interface components and processes in sections 4.4 and 4.5 only covers the core of the application: the input form, the parse browser and the result viewer. In this section we discuss additional components that make the application complete.

Examples Browser

The examples browser displays a list of examples that can serve to exemplify interesting natural language phenomena and to demonstrate the capabilities and limitations of the parser.

Examples are categorized in groups that share the same phenomenon and each example has a hyperlink that takes the user directly to the parse browser for the given sentence.

The page shows a clickable list of phenomena at the top such that the user can quickly jump to examples of a specific phenomenon.

The examples controller loads the examples with the examples processor and processes the examples by adding a link to results controller for each example before having the view render the page.

Examples should also work when the parser server is offline or does not work for some reason. All examples are cached with infinite cache lifetime such that they do not get purged during the periodic cache cleaning.

Help Pages

The help controller has an action for each area that is covered with context sensitive help text. The help area is detected by the base controller. Currently, the controller does nothing and lets the framework render the contents of the corresponding view.

The help text for each help area (or context) is defined in the corresponding view file.

About page

The about page is a XHTML page with information about the application and the parser. Like the help controller, the about controller does nothing. The corresponding view is rendered by the framework.

Setup Routine

When deploying the application, the setup routine has to be run once. The setup routine executes the following operations:

- It asks the grammar processor to validate the grammar.xml file and to cache the parsed data.
- The thesaurus and the full forms lexicon are asked to build and populate their database from the examples.xml and the fullforms.txt files.
- Finally, it asks the examples processor to validate and load all examples from the examples.xml file. Then it uses the result store to parse all examples and to cache them with infinite cache lifetime.

Since the setup process can take several minutes, the routine periodically displays a status update of the progress.

The setup routine is an immediate controller and disables the output buffering (caching) that is performed by the Zend controller component by default. Else the status updates would not be transfered to the client before the whole process is complete.

The installation lock file is an empty file that needs to be placed into the applications root folder before the setup controller can be accessed. This is a security mechanism since the setup process is heavy on server resources and any user can access the setup controller. Ensuring that only users with write access to the filesystem can enable the setup controller avoids any confusion.

A warning is shown on all pages of the application if one forgets to remove the installation lock file again after the setup routine has been executed.

4.6 Parser Application Interface

An introduction to the parser interface and parser specific elements has been given in secion . In this section we discuss the topic in detail.

4.6.1 Parser Interface

The interface is defined as a list of XML-RPC methods with the a specification of the method names and parameters and the data to be returned. All data transfered is encoded in UTF-8.

Parse Request

The parse method is mandatory and must be implemented by the parser.

The XML-RPC method invoked is “parser.parse” and the request comes with a single parameter, the text to be parsed as a string.

The server can reply with XML data that conforms with parses.dtd or status.dtd. It is essential that the document type is specified in the reply such that the application can interpret the XML data either according to parses.dtd or to status.dtd.

Normal parse results should conform with parses.dtd while status.dtd can be sent if there are unknown words or exceptions occurred.

Lexicon Check Request

The lexicon check method is *optional* and does not need to be implemented by the parser. If the method is not implemented, the client still tries to perform lexicon checks, but results will not get displayed.

The XML-RPC method invoked is “parser.lexiconCheck” and the request comes with a single parameter, the text to be checked as a string.

The server can reply with XML data that conforms with status.dtd to list unknown words or signal exceptions.

XML Schema for Parses Replies

The detailed schema can be found in appendix A.2. Here we discuss the function and possible values for each XML element and attribute.

A parse request result consists of a list of “<parse>” elements. Each “<parse>” element must have a node tree representing the parse result and can have additional data structures (features) to store referenced data like feature structures.

A “<node>” element is a recursive data structure and can have zero or more child nodes. Example:

```
<node>
  <node/>
  <node>
    <node>
      <node>
    </node>
  </node>
</node>
```

All nodes but leafs must have a grammar rule identifier which usually implies that the node also has child nodes. The value of the “ruleId” argument should either be an integer number corresponding to a rule identifier in the grammar.xml file. Or it can be a string with of the format “ruleX” where X stands for an integer.

All nodes but the root node must have a grammar rule argument identifier which usually implies that this node has a parent node. The format of the value for the “ruleArgumentId” attribute is “X.Y” or “argumentX.Y” where X is an integer representing the a rule identifier and Y is an integer referencing the argument of rule X at position Y.

The optional “partOfSpeech” argument should have a value that consists of the letters A to Z and _ only. The value is also used to color-code a specific word according to its part of speech. Be sure to update the list of color-codes in Application.css if parts of speech are added or removed.

All nodes should have a “text” attribute with the textual representation of the node. For inner nodes this is the phrase they stand for and for leafs (lexical rules and arguments) this can be the part of a word that the node stands for.

The optional “featuresId” attribute of a node can have any value as long as there is a “<features>” element included in the parse result that has the same value for its own “featuresId” attribute. With the “featuresId” attribute additional data can be associated with any node and the application can display this data alongside the derivation tree.

The optional “<features>” attribute currently takes a list of comma separated key value pairs. The application expects the attribute only for word nodes (the first lexical node in the hierarchy). The format of the value is “key 1=value 1,key 2=value 2,key 3=value 3”. Any number of key value pairs are allowed, comma and the equals symbol are not illegal characters within keys and values. The application uses these features as properties that the parse results can be sorted by.

“<Features>” Element : The “<features>” element is not to be confused with the “features” attribute of the “<node>” element.

A “<features>” element usually represents a feature structure but can also just include unformatted text. In the former case it must have a “<featureStructure>” element as child.

A “<featureStructure>” element has a “type” attribute to give the structure a name and an “index” attribute such that other feature structures can be referenced. The element can have any number of “<attribute>” elements as child and each “<attribute¿” element must have a “name” attribute and a “<featureStructure>” as a child.

XML Schema for Status Replies

The detailed schema can be found in appendix A.3. Here we discuss the function and possible values for each XML element and attribute.

A status message is used to signal exceptions and to return a list of unknown words.

Currently there are two different exceptions that can be communicated: “outOfMemeoryException” and “otherException”. In case of the latter, an exception message must can included.

Unknown words are listed as “<unknownWordException>” elements with a “word” attribute to specify the word.

An empty status reply indicates that all words are known and that there are no problems. Example:

```
<?xml version="1.0"?>
<!DOCTYPE parses SYSTEM "parses.dtd">
<status/>
```

4.6.2 Grammar

The grammar must be specified as a UTF-8 encoded XML document conforming with grammar.dtd (see appendix A.1).

The grammar is a list of phrasal and lexical rules. Each rule must have an identifier which should be referenced in the parse result data. The type ("phrasal" or "lexical") must be specified for each rule. If the rule has a head argument, the identifier of the head argument should be specified as well.

A rule can have rule arguments and each rule argument must have identifier which should be referenced in the parse result data.

Rules and rule arguments can be annotated with a name and optionally with text that is rendered as XHTML. All XML and XHTML entities must be encoded if used as attribute values.

Example:

```
<?xml version="1.0"?>
<!DOCTYPE grammar SYSTEM "grammar.dtd">
<grammar>
  <rule ruleId="rule66" headIndex="1" type="phrasal">
    <ruleArgument ruleArgumentId="argument66.0">
      <annotation name="complement"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument66.1">
      <annotation name="head"/>
    </ruleArgument>
    <annotation name="kopf-komplement-schema-verbal"/>
  </rule>
  <rule ruleId="rule5" type="lexical">
    <ruleArgument ruleArgumentId="argument5.0">
      <annotation name="1"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument5.1">
      <annotation name="2"/>
    </ruleArgument>
    <ruleArgument ruleArgumentId="argument5.2">
      <annotation name="3"/>
    </ruleArgument>
    <annotation name="noun-noun-compound"
                text="&lt;a href=&quot;some URL&quot;&gt;Some anchor
                      text&lt;/a&gt;"/>
  </rule>
  ...
</grammar>
```

4.6.3 Thesaurus Data

The thesaurus must be specified as a UTF-8 encoded text file with the filename thesaurus.txt.

Each line in the file specifies a thesaurus entry. The format for an entry is "<term> : <result>" whereas "<term>" can be any character sequence excluding the newline and the colon character and the format of "<result>" is, using the Backus-Naur form: "<result> ::= <term>(<term>)*".

Example:

```
abblocken : abhalten abwehren hindern sperren zusperren
abbremsen : bremsen
abbruch : abbau abriss
```

The terms mentioned on the left side should not be part of the parser's lexicon while the terms on the right side should be included in the lexicon.

4.6.4 Full Forms Lexicon

The full forms lexicon must be specified as a UTF-8 encoded text file with the filename fullforms.txt.

Each line in the file specifies a single full form. A full form is expected to be a single word, yet all characters but the newline character are allowed.

Example:

```
abarbeit
abarbeite
abarbeiten
abarbeitend
abarbeitende
```

The terms listed in the full forms lexicon should all be part of the parser's lexicon. But false positives are catched since parse requests can also reply with a list of unknown words.

4.6.5 List of Legal Characters

All characters that the parser expects to be in the text to be parsed must be specified in the configuration file of the application.

The format allows to specify ranges of characters with the special character "-", e.g. "A-Z". If the dash character itself is a legal character as well, it must be added to the list escaped with two backslash characters.

Example for characters for a parser of German language:

```
$config['parser.legalCharacters'] = "A-Za-zäöüÄÖÜéÉèÈêÊôÔîÎÀàëËÖöïÏ '\\-*";
```

The star character has no special meaning in the application, but the parser might treat it as a special character. If punctuation characters are allowed, they should be included in this list as well.

4.6.6 List of Tokenizer Characters

All characters that should be interpreted as word tokenizer characters must be specified in the configuration file of the application.

The tokenizers are required to provide the auto completion service and on the client side to omit the last word when performing the lexical check requests.

The format for the list of tokenizer characters is the same as the format for the legal characters, but character ranges are not allowed.

Example:

```
$config['parser.tokenizerCharacters'] = ',.: !?*';
```

4.6.7 List of Examples

The list of examples must be specified as a UTF-8 encoded XML document conforming with examples.dtd (see appendix A.4).

The list is organized in "<phenomenon>" elements as categories of examples. Each phenomenon must have an identifier and a description. An example must have an identifier, a description and a phrase.

A sample for an examples.xml file is shown in section 2.3.6.

4.7 Deployment

The deployment is quite simple and the application has low system requirements. After specifying the requirements, we look at the steps required to install and configure the system and how it should be maintained.

4.7.1 Requirements

Web Server

- Any modern web server (HTTP server)
- PHP 5 running either as a web server module or as CGI
- The PHP configuration must load the "SQLite" extension.
- PHP 5 should be configured with a minimum "memory_limit" of "20M" (20 Megabytes)

Notes:

- The exact required PHP version is 5.1.3 or later.
- The application slightly benefits from Apache's mod_rewrite module to generate shorter URLs but it is not a requirement.
- Ideally (optionally), PHP's "output_buffering" configuration option is disabled such that the progress updates are sent to the browser during the setup before the process is finished.

Web Browser

Any modern web browser can be used. To display the derivation tree, an SVG or VML enabled web browser is required. SVG support can be added to most browsers that do not support it natively with a third party browser plugin, such as the Adobe SVG viewer.

Recommended web browsers:

- Mozilla Firefox in version 1.5 or later
- Micorosft Internet Explorer 6 or later
- Opera 9 or later

Parser Server

The parser server needs to implement the XML-RPC protocol and the serer needs to be reachable from the web server. There are no further requirements.

4.7.2 Installation

To deploy the application the whole application folder needs to be copied to the webserver into a web accessible directory.

The parser specific files in the "application/resources/" subfolder should be replaced.

The parser server can be installed on another machine.

A note on security: Not all subfolders of the application need to be in a web accessible directory. It is sufficient when the two files "index.php" and "config.php" and the subfolder "static" are in the web accessible directory. All other folders can be moved to another location as long as the corresponding configuration paths are updated accordingly.

As long as the web server respects the ",htaccess" directives in the subfolders, one can leave all subfolders in the web accessible directory. There is no sensitive data stored in any of the folders, but one should consider to move the folders out of the web accessible directory if possible to avoid logs or other data to be read from users.

4.7.3 Configuration

Before the application can be configured, the parser server should be started.

The application can be configured with the "config.php" file. The file can be edited with any text editor and should be saved with UTF-8 encoding.

The most important configuration options are:

- "parser.server" specifies the URL where XML-RPC requests should be posted to.
- "parser.legalCharacters" and "parser.tokenizerCharacters" Must be updated to reflect the characters expected by the parser.
- "parser.websiteUrl" and "parser.publicLabel" are shown in the footer of every page as the parser's name and website URL.
- "data.path" is the path to the data folder of the application. The data folder needs to be writeable for the process the web server (or PHP) runs as.
- "log.path" is the path to folder where log files are stored. It needs to be writeable for the proces the web server (or PHP) runs as.

Once the configuration file has been changed and saved, the application can be initialized by running the setup routine.

To access the setup routine a file with the name "INSTALL_LOCK" must be placed into the root folder of the the application.

To run the setup routine, open the URL "http://example.com/parserWebIf/index.php/setup" in a web browser. "example.com/parserWebIf/" is a placeholder and needs to be replaced with the actual values, the domain name and the folder name where the application has been installed.

Once the setup routine reports that it has succeeded, the installation lock file should be deleted again.

Note: The setup routine does not read the thesaurus and full forms lexicon data again if the database already exists. To recreate the database from the data in the files, delete the file "data/database.db" before running the setup routine.

4.7.4 Maintenance

- The log file at "logs/log.txt" should be monitored for warnings or even fatal errors, e.g. "Parser_HttpException" which are thrown when the parser server is unavailable.
- Since the results cache has no fix limit and its size is controlled by deleting results that are older than the configured cache lifetime, the cache can consume a lot of disk space. The cache can be cleared manually by deleting the contents of the "data/cache/" folder.

4.8 Debugging

When modifying the system or debugging an error condition, the following tools can be useful:

- The level at which events are logged can be configured in the application's configuration file. Set the "log.level" to "DEBUG" to log everything that can be logged and to enable XML schema validation for all replies that are received by the parser client.
- Enable the "debug.mode" of the application to see exceptions and their stack trace directly in the browser and to display an input form on the input page to specify the address of the parser server.
- Firebug is highly recommended to debug any code that runs in the web browser. Firebug is a free plugin for the Mozilla Firefox (a web browser) and is an invaluable tool for web developers.
- Use the W3C Validator to validate all generated XHTML pages. All pages generated by the application should be "XHTML 1.0 strict" compliant, else some web browsers switch into "quirks mode" which can result in unexpted effects. Quirks mode is used by some web browsers to maintain backwards compatibility with web pages designed for older web browsers.

Chapter 5

Conclusions and Outlook

5.0.1 Conclusions

A three tier architecture has been implemented with the *web browser* as the user interface, a *web server application* as request handler and transparent cache for parse results and a separate *parser server* to provide a parse service.

The separation of web application and parser server was needed to fulfill the requirements of reusability with different parsers and to account for the different requirements and application profiles of the heavy parser server and light web application.

The parser interface is general enough to use the web interface with any natural language parser. The parser can be written in any programming language, can run on a sepate server and have any requirements. The parser is compatible with the web interface as long as it replies to the series of defined XML-RPC request methods with replies that conform to the defined XML schemas,

Easy deployment has been achieved by using PHP as the programming language for the web application and using its built-in “SQLite” extension for persistence. These technologies are already installed on most web servers and do not need any special configuration. The requirements of the web application have been kept small by applying techniques with little resource usage and by avoiding technologies that are not ubiquitous.

Common visalizations like the derivation tree have been implemented and augmented with some interactive elements.

A concise visualization of readings has been achieved with the disambiguation widget. It shows key properties of a parse result in a very compact way and supports user interaction by highlighting phrasal structure and offering context sensitive sorting criteria.

5.0.2 Outlook

It will be interesting to see whether this web interface for natural language parsers will find adoption in the field of computer linguists and whether it can be reused as easily as expected.

While the web interface offers most features that one would expect, it still lacks an important one and some details could be improved:

- *Feature structures* are not yet processed and visualized by the web interface. It should be fairly simple to add this feature to the application as some of the applications presented in section 1.2 have shown. Feature structures can be rendered using basic XHTML and CSS.

- The *"parses schema"* could be improved by removing the "features" attribute of the "<node>" element since it is redundant with the "<features>" element. It only makes sense to have both if one insists on limiting the features that can be sorted by a subset of all features listed in a feature structure.

- It would also be useful if the XML schema of the allowed XML-RPC replies allowed to produce statistical data. The web application could display the elapsed time to perform the parsing and other data could be aggregated.

- The *cache* of the web application could be refactored to scale better. It stores a list of all parse results in a single cache entry as serialized objects. A single cache entry can be in the order of megabytes. The data that is stored (serialized objects) could be made less verbose and it could be compressed with "gzip". Storing each parse result separately would ensure that a cache entry is a lot smaller than it is now. With a lower requirement for the size of cache entries, one could switch from using a file based cache to an "SQLite" based cache and profit from faster lookups.

- The entries of the thesaurus and the full form lexicon should be stored without converting all text to lower case to enable the best possible user experience.

Bibliography

[1] Ulrich Schäfer, *Heart of Gold – an XML-based middleware for the integration of deep and shallow natural language processing components, User and Developer Documentation*, DFKI Language Technology Lab, Saarbrücken, Germany, 2005.

[2] Ivan A. Sag and Thomas Wasow, *Syntactic Theory: A Formal Introduction*, Center for the Study of Language and Information, January 1999.

[3] I. Cruz and R. Tamassia, "Graph drawing tutorial," .

[4] "Yahoo! ui library: Graded browser support," .

[5] II J. Q. Walker, "A node-positioning algorithm for general trees," *Softw. Pract. Exper.*, vol. 20, no. 7, pp. 685–705, 1990.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., 1995.

Appendix A

XML Schemas

A.1 Grammar.dtd

```
<?xml encoding="UTF-8"?>

<!ELEMENT grammar (rule)*>

<!ELEMENT rule (ruleArgument+,annotation?)>
<!ATTLIST rule
  ruleId CDATA #REQUIRED
  headIndex CDATA #IMPLIED
  type (phrasal|lexical) #REQUIRED>

<!ELEMENT ruleArgument (annotation)?>
<!ATTLIST ruleArgument
  ruleArgumentId CDATA #REQUIRED>

<!ELEMENT annotation EMPTY>
<!ATTLIST annotation
  name CDATA #REQUIRED
  text CDATA #IMPLIED>
```

A.2 Parses.dtd

```
<?xml encoding="UTF-8"?>

<!ELEMENT parses (parse)*>

<!ELEMENT parse (node,features*)>
<!ATTLIST parse
  weight CDATA #REQUIRED>

<!ELEMENT node (node)*>
<!ATTLIST node
```

```
  ruleId CDATA #IMPLIED
  ruleArgumentId CDATA #IMPLIED
  featuresId CDATA #IMPLIED
  partOfSpeech CDATA #IMPLIED
  features CDATA #IMPLIED
  text CDATA #IMPLIED>

<!ELEMENT features (#PCDATA|featureStructure)*>
<!ATTLIST features
  featuresId CDATA #REQUIRED>

<!ELEMENT featureStructure (attribute)*>
<!ATTLIST featureStructure
  type CDATA #REQUIRED
  index CDATA #IMPLIED>

<!ELEMENT attribute (featureStructure)>
<!ATTLIST attribute
  name CDATA #REQUIRED>
```

A.3 Status.dtd

```
<?xml encoding="UTF-8"?>

<!ELEMENT status (outOfMemoryException|unknownWordException|otherException)*>

<!ELEMENT outOfMemoryException EMPTY>

<!ELEMENT unknownWordException EMPTY>
<!ATTLIST unknownWordException word CDATA #REQUIRED>

<!ELEMENT otherException EMPTY>
<!ATTLIST otherException message CDATA #REQUIRED>
```

A.4 Examples.dtd

```
<?xml encoding="UTF-8"?>

<!ELEMENT examples (phenomenon)*>

<!ELEMENT phenomenon (example)*>
<!ATTLIST phenomenon
  id CDATA #REQUIRED
  description CDATA #REQUIRED>
```

```
<!ELEMENT example EMPTY>
<!ATTLIST example
  id CDATA #IMPLIED
  phrase CDATA #REQUIRED
  status CDATA #IMPLIED
  description CDATA #IMPLIED
  comment CDATA #IMPLIED>
```

Appendix B

Task Description

Semesterarbeit für Andres Staudacher

Betreuer: Tobias Kaufmann ETZ D97.7

Stellvertreter: Thomas Ewender ETZ D97.7

Ausgabe: 8. August 2007

Abgabe: Oktober 2007

Entwicklung eines Web-Interfaces für einen natürlichsprachigen Parser

Einleitung

Für jemanden, der sich nicht intensiv mit Natural Language Processing (NLP) auseinandergesetzt hat, ist die Komplexität natürlicher Sprache und die Schwierigkeiten ihrer Verarbeitung kaum vorstellbar. Eine webbasierte Schnittstelle zu einem natürlichsprachigen Parser könnte dem interessierten Laien einen Eindruck von dieser Komplexität vermitteln. Für Experten wiederum kann eine solche Schnittstelle nützlich sein, weil die Funktionsweise einer bestimmten Grammatik in der direkten Interaktion mit dem Parser oft leichter erfasst werden kann als durch das Studium der Grammatik selbst.

Aufgabenstellung

Ziel dieser Semesterarbeit ist es, eine webbasierte Schnittstelle zu einem natürlichsprachigen Parser zu konzipieren und zu implementieren. Mit Hilfe des Betreuers soll zudem ein Demonstrationssystem aufgesetzt werden, wobei der Parser und sämtliche inhaltlichen Informationen dem Studenten zur Verfügung gestellt werden. Es sind folgende Punkte zu berücksichtigen:

Entwicklung einer Benutzerschnittstelle. Die Benutzerschnittstelle sollte so gestaltet sein, dass sie leicht von Laien benutzt werden kann und gleichzeitig den Zugriff auf diejenigen Informationen ermöglicht, die einem Experten nützlich sind. Die wichtigsten Anforderungen können wie folgt umrissen werden:

- Die Schnittstelle soll den Benutzer bei der Eingabe von Sätzen unterstützen. Wünschenswert ist etwa eine Möglichkeit, dem Benutzer die verfügbaren Wörter aufzuzeigen. Auch Sammlungen von kommentierten Beispielsätzen können dazu dienen, den Benutzer auf interessante Phänomene aufmerksam zu machen.
- Da natürlichsprachige Sätze oft mehrdeutig sind, wird der Parser im Allgemeinen mehrere Lesarten entdecken. Die Visualisierung der verschiedenen Lesarten sollte so konzipiert sein, dass sich der Anwender schnell einen Überblick über die Ursachen der Mehrdeutigkeiten verschaffen kann und eine bestimmte Lesart leicht gefunden werden kann.
- Es sollte möglich sein, eine Lesart zu visualisieren, beispielsweise mit Hilfe eines Ableitungsbaums. Wenn möglich sollte die Visualisierung mit Erklärungen und Verweisen (beispielsweise auf erläuternde Publikationen) annotiert werden können.

Entwicklung einer Schnittstelle zum Parser. Die Schnittstelle zum Parser sollte möglichst abstrakt spezifiziert sein, so dass sie für unterschiedliche Grammatiken und Parser verwendet werden kann.

Implementation. Die Implementierung der webbasierten Schnittstelle sollte möglichst portabel, das Deployment möglichst unkompliziert und gut dokumentiert sein.

Die durchgeführten Arbeiten und die erhaltenen Resultate sind in einem Bericht zu dokumentieren (siehe dazu [**?**]), der in gedruckter Form (gebunden) und als PDF abzugeben ist. Zusätzlich sind im Rahmen eines Kolloquiums zwei Präsentationen vorgesehen: etwa eine Woche nach Beginn soll der Arbeitsplan und am Ende der Arbeit das Resultat vorgestellt werden. Die Termine werden später bekannt gegeben.

Literaturverzeichnis

[1] Ulrich Schäfer, *Heart of Gold – an XML-based middleware for the integration of deep and shallow natural language processing components, User and Developer Documentation*, DFKI Language Technology Lab, Saarbrücken, Germany, 2005.

[2] Ivan A. Sag and Thomas Wasow, *Syntactic Theory: A Formal Introduction*, Center for the Study of Language and Information, January 1999.

[3] I. Cruz and R. Tamassia, “Graph drawing tutorial,” .

[4] “Yahoo! ui library: Graded browser support,” .

[5] II J. Q. Walker, “A node-positioning algorithm for general trees,” *Softw. Pract. Exper.*, vol. 20, no. 7, pp. 685–705, 1990.

[6] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley Longman Publishing Co., Inc., 1995.