Abschlussvortrag

Aktualisiertes Abstract

Der Terminparser soll in Freiform-Terminankündigungen (vor allem des Unimut) Datumsangaben, Ort und Zeit von Veranstaltungen taggen. Das Ziel ist es, Termine, die in beliebiger Formulierung eingereicht werden, so aufzubereiten, daß sie bestenfalls sofort ins Netz gestellt oder anderweitig weiterverarbeitet werden können. Dabei sollen bestimmte Felder eben mit Datum, Ort und Zeit der Veranstaltung gefüllt werden. (Zusätzlich gibt es die Möglichkeit, Veranstalter zu taggen, die jedoch eher selten fündig wird.)

Wir parsen auf der Grundlage einer problemspezifischen Grammatik auf XML-Basis die POS-getaggte und teilweise semantisch annotierte Eingabe und gewichten gefundene Objekte nach ihrer in der Grammatik festgelegten Relevanz.

Die Ausgabe wird aus den Objekten mit den höchsten Gewichtungen erzeugt. Sie muss direkt von einem Programmodul ausgewertet werden (da die Darstellung XML nicht entspricht und uns andere Formalismen untauglich erschienen).

Zur Vorbereitung bedienten wir uns eines Korpus von Veranstaltungsankündigungen des Unimut.

Grammatik und Übersetzungsmodule sind erweiterbar und können auf andere Probleme angewendet werden.

Ziele des Projekts

- Extraktion von
  - Ort,
  - Veranstalter,
  - Titel,
  - Zeit und
  - Datum
- aus Freiform- ASCII-Dateien
- plattformunabhängig
- in vernünftiger Zeit

Erreichte Ziele/Zustand

- Extraktion von
  - Datum
  - Zeit und
  - Ort
- aus Freiform-ASCII-Dateien
- in gerade eben erträglicher Zeit

Nicht erreichte Ziele

- Extraktion von
  - Titel und
  - Veranstalter
- plattformunabhängig (diePOS-Tagger-Komponenteläuftz.Zt. nurunterUNIX-Systemen,kannabertheoretischangepasstwerden)
- in vernünftiger Zeit (*knapp* verfehlt)

Nutzen des Projekts

- Unterstützung von überarbeiteten Unimut-Webmastern beim Eintragen von Terminen
- Erweiterbarkeit im Sinne weiterer Extraktionsaufgaben (z. B. liessen sich vermutlich Freiform-Kochrezepte einigermaßen beschreiben, wobei allerdings einige Änderungen am Bewertungssystem nötig wären)

Komponenten

Übersicht

- Tagger
  - POS-Tagger: externer Tagger (TNT) & Integrationsskripte
  - semantischer Tagger: im Wesentlichen lexikonbasiert
- Parser
  - Grammatik-Modul:
    - liest eine XML-Grammatik ein
    - wandelt sie in reguläre Ausdrücke um
    - verwaltet den Zugriff auf diese Ausdrücke (Visitor-Pattern)
  - Text-Modul: interagiert mit dem Grammatik-Modul, eigentlicher “Parser”

Daten

Externe Datenquellen

- Der Originaltext
- Die semantischen Lexika
- Die Grammatik
  - im XML-Format
  - reguläre Sprache (eingeschränkt)

Daten

Text-Beispiel

[...]

Heidelberg, den 20. September 2001

Sehr geehrte Damen und Herren,

wir möchten auf folgende Veranstaltung hinweisen:

Eine Besichtigung des Tiefmagazins der Universitätsbibliothek Heidelberg bietet

der Heidelberger Geschichtsverein e.V. allen Interessierten. Am Dienstag, dem

23. Oktober 2001, führt Veit Probst durch das Magazin unter dem Gebäude der

Neuen Universität und erklärt die Technik. Treffpunkt ist um 18 Uhr im Foyer der

Universitätsbibliothek Heidelberg, Plöck, gegenüber der Peterskirche. [...]

Daten

Ein semantisches Lexikon

Aus germanCities.txt:

```
Aalen
Achern
Adenau
Ahaus
Ahlen
Ahrensburg
Ahrensb&ouml;k
Aichach
```

Daten

Grammatik-Beispiel

```
<xml>
 <vort score="60">
  <lit>Treffpunkt</lit>
  <pos>PER</pos>
  <this>
   <ANYTHING/>
  </this>
  <LINEBREAK/>
 </vort>
 <vzeit score="60">
  <BEGIN/><lit>Beginn</lit><OR/><lit>Abfahrt</lit><END/>
  <pos>PER</pos>
  <this>
   <sem>zeit</sem>
  </this>
 </vzeit>
</xml>
```

Daten

Die Grammatik (1)

- Jeder einen Eintrag umschließende Tag besteht aus seinem Typ (z. B. “vort”) und einem Score (<vort score=“20”>...</vort>)
- Jeder Eintrag besteht aus drei “Feldern”: dem linken Kontext des Fundes, der Beschreibung des Fundes innerhalb eines “this”-Tags (<this>...</this>) und dem rechten Kontext

Daten

Die Grammatik (2)

- Innerhalb eines Eintrags (Kontexte und “this”; *nicht* überkreuzend) sind folgende Elemente erlaubt:
  - <pos>...</pos> - Ein Token mit dem entsprechenden POS-Tag (<pos>AdjAdv</pos>)
  - <sem>...</sem> - Ein Token mit der entsprechenden semantischen Beschreibung (<sem>ort</sem>)
  - <lit>...</lit> - Ein Token mit dem entsprechenden (literalen) Inhalt (<lit>Einlass</lit>)
  - <linebreak/> - Ein Zeilenumbruch *muss* im Originaltext stehen
  - <wordmax>N</wordmax> - Maximal N Tokens
  - <anytoken/> - Irgendeine Anzahl von Tokens (matcht stingily, keine Linebreaks)
  - <anything/> - Irgendetwas; matcht stingily

Daten

Die Grammatik (3)

- Besonderheiten:
  - Zeilenumbrüche sind jederzeit hinter einem Token möglich
  - LIT-Tags können literale reguläre Ausdrücke als anonyme Gruppen enthalten (nur in einfachen Fällen zu empfehlen: `<lit>(?:montags|dienstags|mittwochs|donnerstags|freitags|samstags|sonntags)</lit>`
  - Gruppierung und Veroderung ist nur mittels eines Hacks möglich:
    - <begin/> entspricht dem Beginn einer Klammerung
    - <end/> ihrem Ende
    - <or/> einem Oder

Beispiel: `<begin/><sem>zeit</sem><or/><lit>irgendwann</lit><end/>`

Daten

Die Grammatik (4)

- Die Elemente POS, SEM, LIT, WORDMAX können das folgende Attribut haben:
  - weaklinebreak="true" - ein Linebreak darf nur stingy folgen; macht die Grammatik langsamer, muss deshalb explizit verlangt werden - die Zusammenhänge sind an dieser Stelle sehr komplex, u. a. da zwischen allen Tokens theoretisch (in einer unformatierten ASCII-Datei) Zeilenumbrüche stehen können
- Die Elemente POS, SEM und LIT können das folgende Attribut haben:
  - opt="true" - Dieses Token ist optional

Daten

Zwischenformate

- Text:
  - (erweiterte) Tokenliste
  - TNT-getaggter Text
    - im Originalformat
    - in XML-Format
  - semantisch annotierter und POS-getaggter Text im XML-Format
- Grammatik: reguläre Ausdrücke

Der getaggte Text

Ein getaggter Text sieht beispielsweise folgendermaßen aus:

```
<?xml version="1.0" standalone="yes"?>
<TEXT>
<token pos="NE" sem="">HEIDELBERGER</token>
<token pos="NE" sem="ort">GESCHICHTSVEREIN</token>
<token pos="NN" sem="">E</token>
<token pos="PER" sem="">.</token>
<token pos="CARD" sem="">V</token>
<token pos="PER" sem="">.</token>
<token pos="NE" sem="">HGV</token>
<LINEBREAK/>
<token pos="NN" sem="">Vorstand</token>
<token pos="PER" sem="">:</token>
<token pos="NE" sem="">Hans-Martin</token>
...
</TEXT>
```

Die regulären Ausdrücke...

... und warum man die nicht von Hand schreiben will

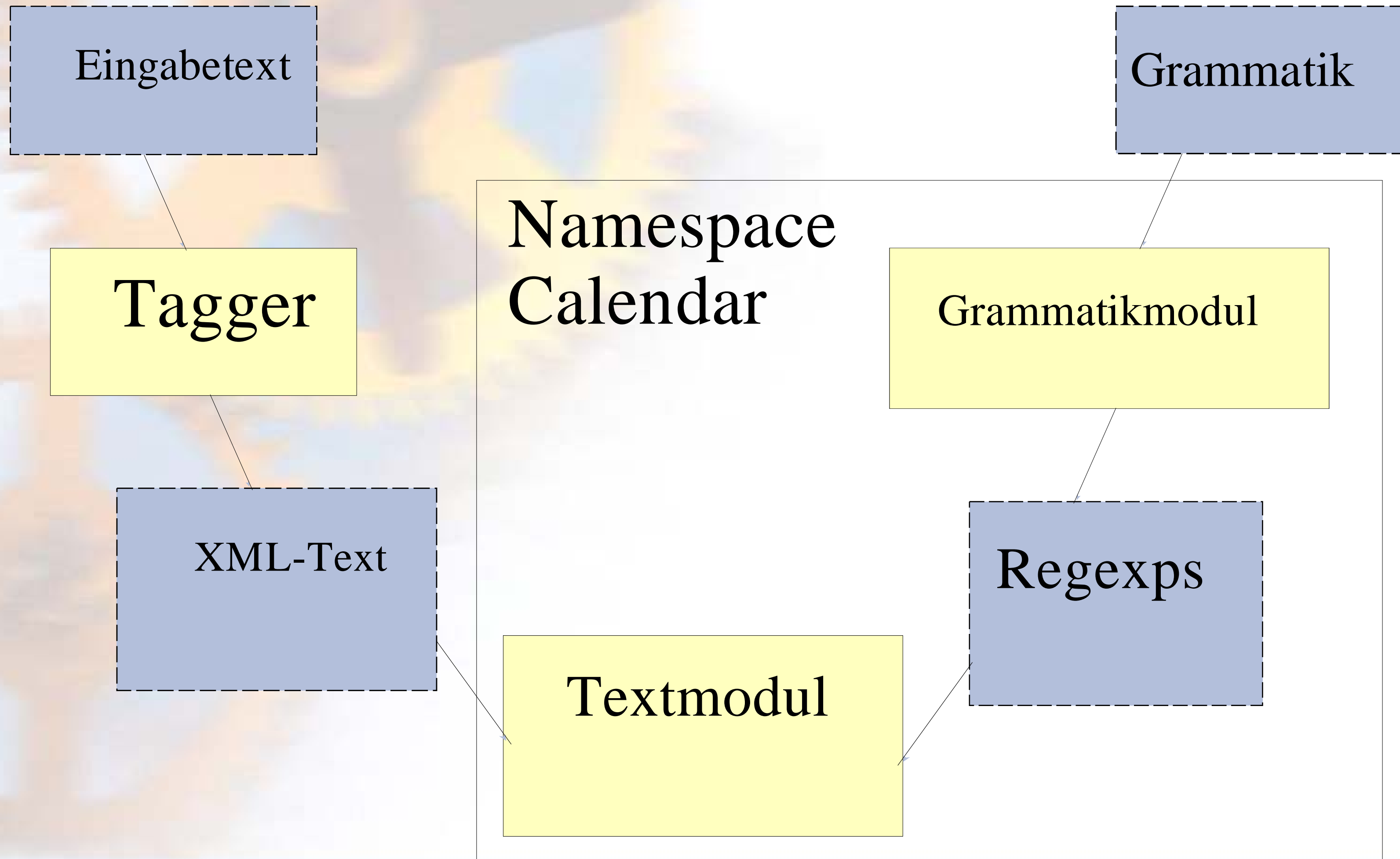
```
<vdat score="55">
<sem>WTAG</sem>
<pos>COM</pos>
<this>
<sem>DATUM</sem>
<sem>JAHR</sem>
</this>
</vdat>
```

aus der Grammatik wird übersetzt in

```
Type: vdat :: Score: 55
Rule: (?mi-
x:((?>\s*<)token\s*(?>[^>]*sem=["']?)WTAG["']?\s*(?>[^>]*>
)[^<>]+\s*<\/token>(?:(?>\s*<)LINEBREAK\/>){0,1}(?>\s*<)to
ken\s*(?>[^>]*pos=["']?)COM["']?\s*(?>[^>]*>)[^<>]+\s*<\/t
oken>(?:(?>\s*<)LINEBREAK\/>){0,1})((?>\s*<)token\s*(?>[^>
]*sem=["']?)DATUM["']?\s*(?>[^>]*>)[^<>]+\s*<\/token>(?:(?
>\s*<)LINEBREAK\/>){0,1}(?>\s*<)token\s*(?>[^>]*sem=["']?)
JAHR["']?\s*(?>[^>]*>)[^<>]+\s*<\/token>(?:(?>\s*<)LINEBRE
AK\/>){0,1})())
```

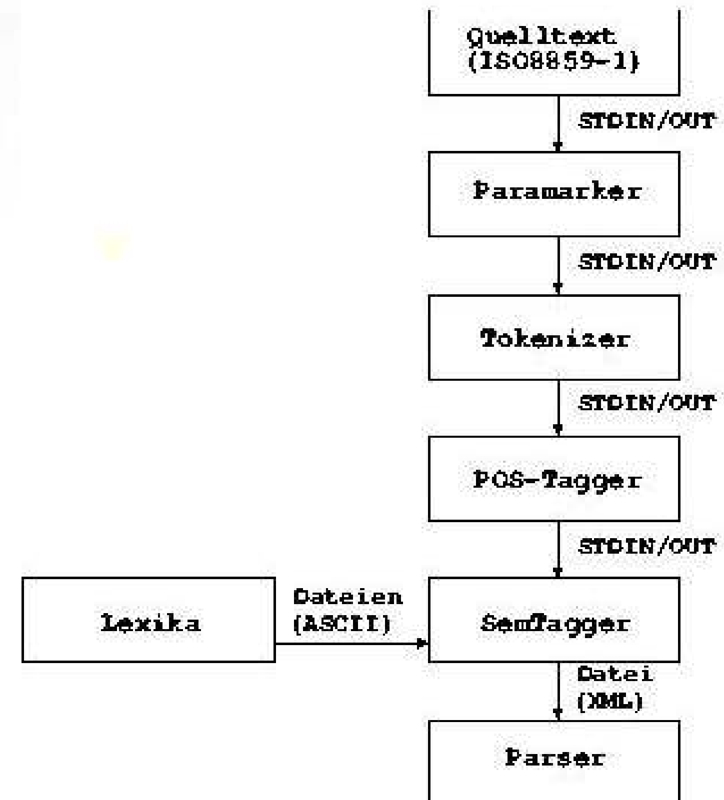
Gesamtsystem

Diagramm



Der Tagger

Ablauf/Diagramm



Der Tagger

Funktionsweise

- Eingabetext wird vorbereitet (Zeilenumbrüche) und tokenisiert
- TNT taggt die Tokenliste
- die TNT-getaggte Ausgabe wird semgetaggt
- die Ausgabe wird umgewandelt in XML-Markup

Der Tagger

Klassen und Methoden, Beschreibungen

- Die Klasse `Word` (nur Attribute):
  - `surface` (String, die Oberflaechenform des Worts)
  - `posTag` (String, die Wortart <- TNT)
  - `semTag` (String, die semantische Kategorie <-get_sem())
  - `index` (Integer, die Position im Text)
- Bsp. f. Hauptfunktionen:
  - get_sem(word, allWords) - gibt (Heuristik) d. sem. Tag zurück
  - merge_words(inWord1, inWord2) - mergt zwei Word-Objekte
- einige I/O - Funktionen:
  - add_pseudo_postags(inText) - taggt alle \n als POS:none
  - clean_xml_string(inText) - ersetzt special chars duch Entities

Das Grammatikmodul

Diagramm

Grammatik

rules.rb
enthält
Übersetzungsregeln

grammar.rb
Klasse: `Calendar::Grammar`
öfftl. Methoden:

```
initialize
precompile filename
each { | r | ... }
```

Das Grammatikmodul

Funktionsweise

- Das Modul kompiliert die Grammatik mithilfe eines XML-Parsers in reguläre Ausdrücke; den Ausdrücken werden Scores und Typen zugeordnet
- Das Modul stellt einen Iterator über sämtliche reguläre Ausdrücke zur Verfügung

Das Grammatikmodul

Klassen und öfftl. Methoden, Beschreibungen

- Klasse `Calendar::Grammar`
  - `initialize`
    - Konstruiert die Klasse
  - `precompile filename`
    - kompiliert eine Grammatik in “filename” m. Hilfe v. rules.rb in reguläre Ausdrücke
  - `each { | r | ... }`
    - iteriert über die regulären Ausdrücke (die regulären Ausdrücke sind im assoziierten Block verfügbar)

Das Textmodul

Diagramm

Grammatik

text.rb

Klasse: `Calendar::Text`

öfftl. Methoden:

```
initialize text, grammar
process
extract type, score=nil
```

Das Textmodul

Funktionsweise

- Das Modul wendet alle regulären Ausdrücke der Grammatik an sämtlichen Positionen (vor jedem Token) an. [... Bug!!! ...]
- Die Funde werden nach dem Parserlauf über eine eigene Methode zur Verfügung gestellt; sie können nach Typ und (Mindest-/Höchst-) Score extrahiert werden.

Das Textmodul

Klassen und öfftl. Methoden, Beschreibungen

- Klasse `Calendar::Text`
  - `initialize text, grammar`
    - Konstruiert die Klasse
    - text ist der (XML-formatierte und getaggte) Eingabetext
    - grammar ist eine Instanz von Calendar::Grammar
  - `process`
    - parst den (XML-formatierten und getaggten) Eingabetext
    - zeitaufwendigster Verabreitungsschritt
  - `extract type, score=nil`
    - extrahiert Funde vom Typ "type", entweder den besten (default) oder alle über score

Probleme

- Geschwindigkeit
- Vergabe von Scores
- Handgeschriebene semantische Kategorien
- Schlechte Beschreibbarkeit bestimmter Zielkategorien, bes. Veranstalter und Titel - wissensintensiv

Probleme

Geschwindigkeit

- Reguläre Ausdrücke können sehr langsam werden
  - Daher müssen Grammatiken von Hand getunt werden
  - Ebenfalls war es sehr zeitaufwendig, den Generator für reguläre Ausdrücke zu tunen
    - "Nested Regexps" erhöhen die Geschwindigkeit teilweise beträchtlich (?>...), weil sie Backtracking verhindern (*Programming Ruby*, S. 210)
    - Stingy reguläre Ausdrücke müssen soweit wie möglich vermieden werden; die meisten stingy matches müssen in der Grammatik explizit angekündigt werden (ausser ANY*)

Probleme

Vergabe von Scores

- Die Vergabe von Scores ist relativ willkürlich-intuitiv
- Sie nimmt einige Zeit in Anspruch
- Sinnvoll wäre es, dies maschinell zu lösen

Probleme

Semantische Kategorien

- Semantische Kategorien müssen von Hand geschrieben werden
- Sie sind nur in sehr geringem Umfang überhaupt vorhanden

Probleme

Mögliche Lösungen: Ausblick (1)

- Geschwindigkeit: eine Reimplementierung in C hätte nur wenige Vorteile; sinnvoll wäre es, den Algorithmus zum Matchen grammatischer Regeln auf den Text zu verbessern (?)
- Scores könnten eventuell mit einem vernünftig großen Korpus maschinell gelernt werden
- Semantische Kategorien ließen sich *möglicherweise* aus GermaNet extrahieren (allerdings eventuell *zu* reich)

Probleme

Wissenslastigkeit; Ausblick (2)

- Es ist z. T. sehr viel Wissen nötig, um bestimmte Kategorien zu beschreiben
- Dieses Wissen ließe sich möglicherweise besser aus einem (*sehr großen*) Korpus lernen
- Eine Möglichkeit zur Verbesserung des Projekts wäre also die Beschaffung und (beliebig reiche) Annotation eines großen Korpus und die Anwendung eines maschinellen Lernverfahrens