

Verwendung der ST Templatesprache zur Präsentation von Treebankdaten

Patrick Gotthardt

patrick@pagosoft.com

Alexander Kornrumpf

alexkornrumpf@web.de

Lara Kresse

lara@larakresse.de

Abstract

Diese Arbeit untersucht die Möglichkeiten der Templatesprache ST in Hinblick auf die Präsentation linguistischer Treebankdaten. Es wird ein einfacher Parser erarbeitet, der Treebankdaten in eine kanonische Datenstruktur aufbereitet. Die Sprache ST wird kurz vorgestellt und anhand verschiedener Präsentationsformate auf die einzelnen Aspekte und Probleme bei der Aufbereitung linguistischer Daten eingegangen. Das Material wird ergänzt durch ein Webbasierendes Programm zur Konvertierung von Treebankdaten.

1 Einführung: Das Penn Treebank-Format

Die Penn Treebank ist eine vielgenutzte Datenbank, die an der University of Pennsylvania gepflegt wird und Part-of-Speech Annotationen englischer Sprachdaten enthält (Marcus et al., 1993). Sie benutzt ein einfaches Klammerungsformat zur Gliederung der Daten. Konstituenten sind jeweils mit runden Klammern „()“ umschlossen. Das direkt auf die öffnenden Klammer folgende Symbol ist die Annotation des Mutterknotens, die darauf folgenden Elemente bis zur schließenden Klammer bilden ggf. Subkonstituenten und sind intern nach demselben Schema aufgebaut.

Beispiel

```
( S ( NP-SBJ ( NP ( NNP Pierre ) ( NNP
  Vinken ) ) ( , , )
  ( ADJP ( NP ( CD 61 ) ( NNS years ) ) ( JJ
    old ) ) ( , , ) ) )
```

```
( VP ( MD will ) ( VP ( VB join ) ( NP (
  DT the ) ( NN board ) ) ( PP-CLR ( IN
  as ) ( NP ( DT a ) ( JJ nonexecutive )
  ( NN director ) ) ) ( NP-TMP ( NNP
  Nov. ) ( CD 29 ) ) ) ) ( . . ) )
```

Das Penn Treebank-Format ist einfach maschinell zu erstellen und zu verarbeiten, es ist jedoch, wie das Beispiel zeigt, für den menschlichen Leser nicht unbedingt übersichtlich zu lesen. Im Rahmen dieser Arbeit soll der erstgenannte Vorteil ausgenutzt werden um eine übersichtlichere Präsentation automatisiert zu generieren. Es ist zu untersuchen, inwieweit die ST Templatesprache zu diesem Zweck geeignet ist.

2 Ein einfacher Treebank-Parser

Der erste Schritt zur Verarbeitung von Treebankdaten in PHP ist das Parsing von flachem Text in eine geeignetere Datenstruktur. Hierzu wird jedes Klammerpaar als Begrenzung eines Arrays aufgefasst. Um die Arrays aus dem Text zu erzeugen, wurde das PgsParser Framework verwendet, das sich am (ANTLR3, 2007) Framework orientiert.

Listing 1: parse(\$input)

```
public static function parse($input) {
    $lexer = new Pgs_Treebank_Lexer(new
        Pgs_Parser_StringStream($input));
    $parser = new Pgs_Treebank_Parser(new
        Pgs_Parser_TokenStream($lexer));
    return $parser->matchExprs();
}
```

Die Parsingfunktion greift auf zwei Komponenten zu, den Lexer und den Parser. Der Lexer führt eine lexikalische Analyse durch und teilt den Text in

Token auf. Das Treebank-Format kennt drei Arten von Token: öffnende Klammer (LPAREN), schließende Klammer (RPAREN) und Bezeichner(ID).

Listing 2: mTokens()

```
public function mTokens() {
    if ($this->isWhitespace($this->input->
        LA(1))) {
        $this->matchWS();
    } elseif (!$this->matchSymbols($this->
        symbolMatches)) {
        $this->type = self::ID;
        while (($c = $this->input->LA(1)) !=
            Pgs_Parser_StringStream::EOF) {
            if ($this->isWhitespace($c) || $c ==
                '(' || $c == ')') break;
            else $this->input->consume();
        }
    }
}
```

Die Tokenizerfunktion prüft zunächst ob es sich bei dem nächsten Zeichen um ein `Whitespace` handelt (`LA(1)`) und `matcht` es gegebenenfalls. Anderenfalls wird zunächst versucht eines der Symbole (und) direkt zu `matchen`. Sollte auch dies nicht gelingen, handelt es sich um einen Bezeichner, der verarbeitet wird.

Listing 3: matchExpr()

```
public function matchExpr() {
    $this->match(self::LPAREN);
    $tree = array();
    $tree[] = $this->match(self::ID)->
        getText();
    while ($this->input->LA(1) != self::
        RPAREN) {
        if ($this->input->LA(1) == self::ID)
            {
                $tree[] = $this->match(self::ID)->
                    getText();
            } elseif ($this->input->LA(1) == self
                ::LPAREN) {
                $tree[] = $this->matchExpr();
            } else {
                $this->noViableAlternative($this->
                    input->LA(1), array(self::
                        LPAREN, self::ID));
            }
    }
    $this->match(self::RPAREN);
    return $tree;
}
```

Der Parser teilt den Baum rekursiv in seine Unterbäume auf. Zunächst wird das einleitende `LPAREN` gematcht und ein Array erstellt, das den Teilbaum enthalten soll. Der Bezeichner des Teilbaums wird

als erstes Element an dieses Array angehängt. Wenn es sich um einen präterminalen Knoten handelt, ist die nächste Subkonstituente kein Baum sondern genau der terminale Knoten. Anderenfalls ist die Subkonstituente wieder ein Baum und muss mit `LPAREN` beginnen. Trifft auch dies nicht zu, so ist der geparsete Ausdruck nicht im Treebank-Format und eine Exception wird geworfen.

Welcher der drei Fälle zutrifft wird mit einem Look-Ahead von 1 geprüft. Die Subkonstituenten werden an das Array angehängt, wobei Teilbäume rekursiv gematcht werden und ihrerseits Arrays sind. Der Vorgang wird solange wiederholt bis der Look-Ahead ein abschließendes `RPAREN` aufspürt. Dieses wird dann gematcht und der komplette Teilbaum als Array zurückgegeben.

3 Die ST Templatesprache

Die so gewonnen Baumdaten sollen nun auf verschiedene Weise präsentiert werden. Hierzu wird `ST` verwendet. `ST` ist eine Sprache um Templates zur Datenpräsentation zu definieren. `ST` wird nach `PHP` kompiliert und kann Funktionen und Klassen aus `PHP` importieren und verwenden.

Templates bestehen aus Funktionen, die Argumente übergeben bekommen und Text zurückgeben. Der Rumpf einer Funktion kann Text, der direkt übernommen wird, und Ausdrücke, die vor der Rückgabe ausgewertet werden, enthalten. Arrayausdrücke werden dabei automatisch reduziert. Die `Map-Operation` erlaubt es jedes Element vor der Reduktion umzuwandeln, optional kann auch ein Trennzeichen festgelegt werden, das zwischen die Elemente eingefügt wird.

Ein Template das Baumdaten als `HTML-Liste` präsentiert soll als einfaches Beispiel zur Erklärung dienen. Hier zunächst das Template im gesamtzusammenhang, im Folgenden wird dann detailliert auf die einzelnen Teile eingegangen.

Listing 4: ListFormatter

```
template ListFormatter;

import Pgs_Treebank_Util [first, rest];
import functionSpace [is_array];

private concat(parts*) = <<$parts$>>
```

```

format(tree) = <<
$tree:concat("<ul>", formatSubtree(_), "</
ul>")$
>>

private formatSubtree(tree) = <<
<li>
$if is_array(tree)$
    $first(tree)$
    <ul>$rest(tree):formatSubtree()$</ul>
$else$
    $tree$
$endif$
</li>
>>

headInclude() = <<
<style type="text/css">
    .category {
        color: #000052;
        font-weight: bold;
    }

    .output {
        font-family: Consolas, "Courier New",
        Courier, Fixed;
        font-size: 14px;
    }
</style>
>>

```

Ein Template beginnt mit der Definition des Template Namens nach dem Schlüsselwort `template`.

Listing 5: ListFormatter

```
template ListFormatter;
```

Mit dem Schlüsselwort **import** können Funktionen aus anderen Klassen und dem PHP `functionSpace` importiert werden. Die Namen der zu importierenden Funktionen stehen dabei Kommasepariert in [].

Listing 6: ListFormatter

```
import Pgs_Treebank_Util [first, rest];
import functionSpace [is_array];
```

Funktionen haben die Form `[private | protected | public] bezeichner([Argumente]) = << Funktionsrumpf >>`. Das letzte Argument einer Funktion kann mit `*` gekennzeichnet werden, und darf dann beliebig oft wiederholt werden. Intern wird es wie ein Array gehandhabt und daher auch automatisch reduziert. Das `$`-Zeichen grenzt im Funktionsrumpf auswertbare Ausdrücke von konstantem Text ab.

Mit diesen Mitteln ist es sehr einfach eine `concat` Operation zu definieren. Die Funktion übernimmt beliebig viele Argumente und reduziert diesen Array-Ausdruck zu einem flachen String.

Listing 7: ListFormatter

```
private concat(parts*) = <<$parts$>>
```

Eine Alternative zur einfachen Reduktion ist die `Map-Operation` `:`. Hierbei bedeutet `$a:b()$` für jedes Element `_` in `a` wende `b` auf `_` an und reduziere das Array der Ergebnisse. Wenn `b` mehr als ein Argument fordert, kann `partial Application` verwendet werden oder wie im folgenden Beispiel - explizit angegeben werden. Insbesondere können auch Funktionen und mit `"` begrenzte Strings als Argumente übergeben werden.

Diese Funktion beginnt und schließt für jeden vollständigen Baum im Array eine neue HTML-Liste und formatiert den Baum entsprechend der Regeln von `formatSubtree`.

Listing 8: ListFormatter

```
format(tree) = <<
$tree:concat("<ul>", formatSubtree(_), "</
ul>")$
>>
```

Enthält eine Funktion konstanten Text, so wird dieser direkt von der Funktion zurückgegeben. Zum Beispiel die folgenden `css` Angaben.

Listing 9: ListFormatter

```
headInclude() = <<
<style type="text/css">
    .category {
        color: #000052;
        font-weight: bold;
    }

    .output {
        font-family: Consolas, "Courier New",
        Courier, Fixed;
        font-size: 14px;
    }
</style>
>>
```

Innerhalb des Funktionsrumpfes können auch Ausdrücke und Text vermischt werden, importierte Funktionen aufgerufen und Ausdrücke mit `if` `Selse$` `Sendif$` geprüft werden.

Die Funktion `formatSubtree` ist das eigentliche Herzstück der Formatierung. Jeder Knoten ganz gleich ob terminal oder nicht wird in ein HTML list item eingebettet. Mit `$if` und der PHP Funktion `is_array` wird geprüft ob es sich um einen Teilbaum oder um einen terminalen Knoten handelt. Terminale Knoten werden im `$else`-Zweig mit ihrem Namen beschriftet. Teilbäume werden in ihre Mutterknoten (`first (tree)`) und Subkonstituenten (`rest (tree)`) aufgespalten. Die Subkonstituenten werden in einer eingebetteten HTML Liste rekursiv nach denselben Regeln formatiert.

Listing 10: ListFormatter

```
private formatSubtree( tree ) = <<
<li>
  $if is_array( tree )$
    $first( tree )$
    <ul>$rest( tree ):formatSubtree ()$</ul>
  $else$
    $tree$
  $endif$
</li>
>>
```

4 Präsentationsformate

4.1 Textformate

Ganz analog zu Listenbeispiel lassen sich beliebige weitere Textformate definieren. Im Rahmen dieses Projektes wurden zwei Beispiele implementiert.

1. Das Bracket-Format. Dieses Format ähnelt in gewisser Weise dem Penn Treebank-Format. Im Unterschied dazu werden aber nur die Markierungen der Terminalknoten ausgeschrieben, während die anderen Markierungen als Subskript an eckige Klammern „[“ angehängt werden. Auf diese Weise bleibt der ursprüngliche Satz im Gegensatz zum Penn Treebank-Format lesbar.

Listing 11: BracketFormatter

```
private formatSubtree( tree ) = <<
$if is_array( tree )$
  <span class="symbol">[ </span><sub>
    $first( tree )$</sub> $rest( tree ):
    formatSubtree ()$<span class="
    symbol">]</span>
$else$ <span class="terminal">$tree$ </
span> $endif$
>>
```

2. Pretty-Print. Während das Bracket-Format im Vergleich zum Penn Treebank-Format den ursprünglichen Satz betont, wird bei diesem Format durch Einrückungen die Baumstruktur betont. Knoten werden abhängig von ihrer Entfernung zum Mutterknoten eingerückt. Dieses Beispiel zeigt einen rekursiven Aufruf mit der Tiefe als weiterem Parameter.

Listing 12: PrettyFormatter

```
private formatSubtree( tree , depth ) =
<<
$! break to next line if this is not
the very first sentence !$
$if depth!=0$<br />$endif$
$! indent properly !$
$str_repeat( "&nbsp;" , depth*4 )$
<span class="symbol"></span> <span
class="category">$first( tree )$</
span>
$! Is the second element a node? !$
$if count( tree ) == 2 && false == is_
array( tree [ 1 ] )$
  <span class="terminal">$tree [ 1 ]$</
span>
$else$
  $! otherwise break and start a new
line !$
  $rest( tree ):formatSubtree( _ , depth
+1 )$
$endif$
<span class="symbol">></span>
>>
```

4.2 SVG

SVG (Scalable Vector Graphics) ist ein XML basiertes Dateiformat zur Beschreibung von Vektorgraphiken, das u.a. für den Einsatz im Web entworfen wurde (W3C, 2004). Leider sind noch nicht alle Browser in der Lage embedded SVG zu rendern. Insbesondere mit dem Internet Explorer kann es zu Problemen kommen. Für beste Ergebnisse wird der Opera-Browser empfohlen.

Trotz der Probleme mit einzelnen Browsern, erscheint SVG als sinnvolle Wahl für die „on-the-fly“ Generierung von Graphiken für das Web. Es ist anzunehmen, dass die Bedeutung und Unterstützung von SVG in der Zukunft weiter zunehmen wird. Daher soll SVG hier als Format für die graphische Präsentation von Treebank-Daten dienen.

Die graphische Präsentation unterscheidet sich von der textuellen fundamental dadurch, dass sie nicht linear ist. Die Zeichenposition ist jederzeit frei

auf einer 2-Dimensionalen Fläche wählbar und muss zunächst bestimmt werden. Die Durchführung dieser Berechnung ist dadurch erschwert, dass ST nativ keine Variablen und keine Rückgabewerte von einem anderen Typ als String unterstützt. Diese Sprachmittel müssen durch importierte PHP Funktionen emuliert werden. Zu diesem Zweck dienen zwei Klassen, die eine Statusverwaltung und einen Stack jeweils mittels eines Arrays implementieren.

Listing 13: PgsTreebankState

```
class Pgs_Treebank_State {
    private $data;

    public function set($name, $value) {
        $this->data[$name] = $value; }
    public function get($name) { return
        $this->data[$name]; }
    public function has($name) { return
        isset($this->data[$name]); }
    public function clear() { $this->data =
        array(); }
}
```

Listing 14: PgsTreebankStack

```
class Pgs_Treebank_Stack {
    private $stack = array();

    public function push($val, $stackName='_
    _DEFAULT_') {
        if(!isset($this->stack[$stackName]))
            $this->stack[$stackName] = array();
        $this->stack[$stackName][] = $val;
    }
    public function pop($stackName='_
    _DEFAULT_') {
        return array_pop($this->stack[
        $stackName]);
    }
}
```

Ein einfaches SVG Format, das den Baum horizontal darstellt, ist dem Pretty-Print Format sehr ähnlich. Zunächst werden zwei „Konstanten“ w und h gesetzt, welche die Breite und Höhe eines Knotens festlegen. Die x-Koordinate eines Knotens ist nun analog zum Pretty-Print gerade `depth*get("h")`.

Zur Bestimmung der y-Koordinate werden immer die Kindknoten vor den Elternknoten gerendert. Dabei wird die Anzahl der bisherigen Terminalknoten festgehalten. Terminale und präterminale Knoten können direkt an der Position `get("tCount")*get("w")` gerendert und verbunden werden.

Für alle anderen Knoten wird bestimmt, wieviele terminale Knoten unter dem zu zeichnenden Knoten liegen und der Knoten mittig über diese gesetzt. Die direkten Kindknoten sind beim Rendern auf einem Stack abzulegen um sie mit den Elternknoten verbinden zu können.

Listing 15: SVGHFormatter

```
svg(tree, depth) = <<
    if is_array(tree[1])$

        $! draw children first !$
        $push(get("tCount"))$
        $rest(tree):svg(, depth+1)$

        $! now place this node at (tCount-pop()
        /2 !$
        $set("pop", pop())$
        $set("x", depth*get("h"))$
        $set("y", ( get("pop") + (1+get("tCount"
        )-get("pop"))/2 ) * get("w"))$

        $rest(tree):paintLine(get("x"),get("y"))
        $

    <svg:text x="$get("x")$px" y="$get("y")
    $px">$tree[0]$</svg:text>

    $push(get("x"), "nodes")$
    $push(get("y"), "nodes")$

$else$
    $!Terminal: $tree; separator="/"$ (x =
    $get("tCount")$, y=$depth!)$

    $set("tCount", get("tCount")+1)$

    $push(depth*get("h"), "nodes")$
    $push(get("tCount")*get("w"), "nodes")$

    $!render preterminal!$
    <svg:text y="$get("tCount")*get("w")$px"
    x="$depth*get("h")$px">$tree[0]$</
    svg:text>

    $!render terminal!$
    <svg:text y="$get("tCount")*get("w")$px"
    x="$depth*get("h")$px">$tree[1]
    $</svg:text>

    $!connect terminal and preterminal!$
    <svg:line x1="$depth*get("h")+25$px" y1=
    "$get("tCount")*get("w")-5$px"
    x2="$depth*get("h")$px" y2="$get("
    tCount")*get("w")-5$px"
    style="stroke:rgb(99,99,99);stroke-
    width:1"/>

$endif$
>>
```

Für eine vertikale Darstellung müssen bis auf kosmetische Änderungen im Wesentlichen nur die x- und y-Koordinaten vertauscht werden. Solche Bäume werden allerdings sehr breit. Es wurde daher die Skalierbarkeit von SVG ausgenutzt um eine Vorschauversion und eine Version in Originalbreite anbieten zu können.

5 Fazit und kritische Würdigung

Die empirischen Untersuchungen haben gezeigt, dass ST sehr gut geeignet ist um Treebank-Daten in verschiedenen Formaten für das Web aufzubereiten. Besonders bei textbasierten Formaten erlaubt die Sprache sehr kurzen und übersichtlichen Code.

Bei Graphikformaten wurden die Betrachtungen auf ein ebenfalls textbasiertes Beschreibungsformat reduziert, für das Erstellen von Binärdateien ist ST nicht ausgelegt. Es hat sich gezeigt, dass mit ST auch sehr zufriedenstellende SVG erzeugt werden können, wobei in dieser ersten Arbeit noch nicht alle Möglichkeiten von SVG ausgeschöpft wurden.

Trotz der ermutigenden Ergebnisse, ist die Durchführung von Berechnungen in ST kritisch zu sehen. Einerseits wurde bewiesen, dass ST durch den import von PHP Funktionen sehr mächtig ist, andererseits lenkt der Code zur Berechnung von Positionen vom eigentlichen Zweck, der Präsentation von Daten, ab. Hier ist es möglicherweise zu bevorzugen, die Baumdaten schon im PHP Code mit Strukturinformation zu versehen.

Eine Webanwendung, welche die Ergebnisse dieser Arbeit demonstriert kann unter <http://www.pagosoft.com/~pago/treebank/index.php> abgerufen werden. Jenseits des Web sind weitere Formate wie z.B. Latex als Anwendung denkbar. Der Einsatz von ST ist aber vor allem Empfehlenswert bei Präsentationsformaten. Für die Umwandlung in andere Datenformate, z.B. (Tiger-XML, 2003), ist ST aufgrund der genannten Einschränkungen in Bezug auf Variablen und Rückgabewerte noch nicht die erste Wahl.

References

- ANTLR, ANOther Tool for Language Recognition Online Erhältlich: <http://www.antlr.org/>
- Mitchell P. Marcus, Beatrice Santorini, Mary Ann Marcinkiewicz 1993. *Building a large annotated corpus of English: the Penn Treebank* Online Erhältlich: <http://www ldc.upenn.edu/Catalog/docs/LDC95T7/c193.html>
- The TIGER-XML treebank encoding format Online Erhältlich: <http://www.ims.uni-stuttgart.de/projekte/TIGER/TIGERSearch/doc/html/TigerXML.html>
- Chris Lilley, Dean Jackson 2004. *About SVG: 2d Graphics in XML* Online Erhältlich: <http://www.w3.org/Graphics/SVG/About.html>