Making sense of legal texts

de Maat, E.

Publication date
2012

Appendix A Detecting Structure Using JAPE

In chapter 2, a method was outlined for the detection of the structure of Dutch legislative documents. This method has been used to construct a prototype using GATE (General Architecture for Text Engineering)[182]. In this appendix, we'll describe the different elements of the JAPE regular expressions that make up this prototype.

A.1 A Short Introduction to JAPE

In GATE, JAPE regular expressions are used to describe patterns that we want to recognise. Before applying such patterns, the document is first split into tokens by the tokeniser. Words, numbers and punctuation are marked using tokens, and whitespace is marked using a spacetoken. We can specify patterns using these tokens.

The most basic pattern is: {*Token*}, which will match any word, number or punctuation. More specific patterns can be created by checking for certain properties of the token. The most commonly used property is the string property, which is equal to the actual word, number etc. For example, if we want to search for the word *Netherlands*, the appropriate pattern is {*Token.string == "Netherlands"*}.

Tokens may be combined to form more complicated patterns. For example, we can search for the phrase *of the Netherlands* using the pattern {*token.string == "of"*} {*SpaceToken*} {*Token.string == "the"*}{*SpaceToken*} {*Token.string = ="Netherlands"*}. If we are not interested in the whitespace, we can instruct the pattern recogniser to ignore the whitespace, in which case the pattern {*token.string == "of"*} {*Token.string == "the"*} {*Token.string == "Netherlands"*} would suffice.

Using specific operators, more complicated patterns may be created:

- A question mark after a pattern enclosed by round brackets indicates that that pattern is optional; it may appear once or not at all.
- A plus-sign after a pattern enclosed by round brackets indicates that that pattern may appear multiple times.
- An asterisk after a pattern enclosed by round brackets indicates that that pattern may appear multiple times and is optional.

To generate output (i.e. to add our own annotations) we can add instructions to each pattern. These instructions are given in the JAVA programming language. To make it easier to add the output, we can add temporary labels to the pattern. For example ({*token.string == "of"*} {*Token.string == "the"*} {*Token.string == "Netherlands"*}):*DutchLabel* would search for the string *of the Netherlands* and then add the temporary label *DutchLabel* to it, which we could later replace by an actual annotation.

A complete JAPE grammar may consist of multiple regular expressions. Furthermore, these expressions may be organised in different phases, which are applied after each other. This means that annotations that are made in the earlier phases can be used as input for later phases.

---

[182] http://gate.ac.uk/

We'll discuss two common patterns used for the determining the structure of laws: the patterns to detect a header, and patterns to detect a fixed element.

A.2 Detecting Headings

As was discussed in chapter 2, a header consists of three elements: a category label, an index and optionally, a title.

The pattern for the category label is simply that label, e.g. for an article header, the pattern for the category label is:

```
{Token.string == "Artikel"}
```

If we assumed that the index can be any number or a combination of numbers, dots and letters that has been identified by the appropriate JAPE grammar, then the pattern for the index is[183]:

```
{Token.kind == "number"}|{Token.kind == "index"}
```

Finally, the title will start with a word which may be followed by a combination of words and punctuation, making the pattern for the title:

```
{Token.kind == "word"}
(({Token.kind == "punctuation"}|{Token.kind == "word"}))*
```

The beginning and the end of the header have been determined by the sentence splitter, meaning that we can expect a split at the beginning and the end of the header. Also, between the index and the title, there may be a dot or a dash. If we add those, combine the individual patterns, and add labels, we get the following complete pattern:

```
Rule: TitledArticleRule
(
      {Split.kind == "external"}
      ({Token.string == "Artikel"}):CategoryLabel
      ({Token.kind == "number"}|{Token.kind ="index"}):IndexLabel
      (({Token.string == "."}|{Token.string == "-"}))?
      (
            {Token.kind == "word"}
            (({Token.kind == "punctuation"}|{Token.kind == "word"}))*
      ):TitleLabel
      {Split.kind == "external"}
) -->
:CategoryLabel.Category = {},
:IndexLabel.Index = {},
:TitleLabel.Title = {}
```

The lines after the arrow (-->) assign annotations to the sections marked by label. The section marked with CategoryLabel is annotated with the annotation Category, etc. Similar rules can be made for other formats (headers without titles, headers using ordinals, etc.)

---

[183] Tokens of kind "index" are not introduced by the GATE tokeniser, but are the result of applying the JAPE grammar that combines numbers/dots/letters.

A.3 Detecting Fixed Elements

As was mentioned in chapter 2, the introduction of a Dutch law starts with *We* and ends with *understands as follows*. This corresponds with the following pattern in JAPE (in Dutch):

```
{Token.string == "Wij"}
({Token})*
{Token.string == "verstaan"}
{Token.string == "bij"}
{Token.string == "deze"}
{Token.string == ":"}
```

A disadvantage of using this pattern is that it may find multiple matches. As the word *We* will probably occur multiple times in the introduction, JAPE may match this pattern to each region that starts with an occurrence of *We* and ends at *verstaan bij deze*. We can instruct JAPE to make only one match. In that case, he will take the match that is the longest, starting at the first *We* and ending at the last *verstaan bij deze*. So, as long as *verstaan bij deze* occurs nowhere else in the document, this is a safe method. This seems like a reasonable assumption.

However, if we do not want to make this assumption, we can get our result by using temporary annotations, by first marking up the first *We* and the first *verstaan bij deze* and then creating a single annotation spanning them.

This requires three phases, two to make the temporary annotations and one to create the final one. The first phase looks as follows:

```
Phase: IntroductionTemporaryAnnotations
Input: Token
Options: control = once

Rule: Wij
(
      {Token.string == "Wij"}
):Wij -->
:Wij.Wij = {}
```

These lines are somewhat more complicated as the pattern shown earlier, as they do not only include the actual pattern, but also all the options and the lines generating the output. We start with

```
Phase: IntroductionStartTemporaryAnnotation
Input: Token
Options: control = once
```

The first line indicates that the patterns that follow belong to a separate phase in the annotation process. The next line indicates that we will only look at Tokens (which means that we will ignore Spacetokens, which actually means that we are not interested in whitespace). The final line indicates that this phase exit after one pattern has been matched – which should be the first occurrence of *We*.

Then, we get the actual rule, which reads:

```
Rule: IntroductionStartRule
(
      {Token.string == "Wij"}
):StartLabel -->
:StartLabel.IntroductionStart = {}
```

The first line indicates that a new rules starts, named *IntroductionStartRule*. Then we get the actual pattern, which consists of a single token matching the word *Wij*. This area is assigned the label *StartLabel*. The final line generates the output: it indicates that an annotation *IntroductionStart* should be applied to the area marked with the label *StartLabel*.

After detecting the first *We* in this manner, we can move on to detecting *understands as follows*, which results in a similar phase:

```
Phase: IntroductionEndTemporaryAnnotation
Input: Token
Options: control = once

Rule: IntroductionEndRule
(
      {Token.string == "verstaan"}
      {Token.string == "bij"}
      {Token.string == "deze"}
      {Token.string == ":"}
):EndLabel -->
:EndLabel.IntroductionEnd = {}
```

Now, we can add a pattern that combines these two temporary annotations to the actual annotation marking the entire introduction.

```
Phase: Introduction
Input: IntroductionStart IntroductionEnd
Options: control = once

Rule: IntroductionRule
(
      {IntroductionStart}{IntroductionEnd}
):IntroductionLabel -->
{
      gate.AnnotationSet as =
         (gate.AnnotationSet)bindings.get("IntroductionLabel");
      outputAS.add(as.firstNode(), as.lastNode(), "Introduction",
        Factory.newFeatureMap());
      outputAS.removeAll(as);
}
```

The pattern links the start and end annotations for the introduction. As the parser has been instructed to ignore everything except the introduction start and the introduction end, we do not need to specify what other tokens may occur in between the start and end of the introduction. The lines that generate the output are a bit more complicated than in the previous rules. This is because we used a shorthand notation in the previous rules. However, we do want to remove the temporary annotations, which cannot be done is shorthand.

The first line of the output generation is:

```
gate.AnnotationSet as =
   (gate.AnnotationSet)bindings.get("IntroductionLabel");
```

This line asks for a set of all annotations that appear in the area that has been labelled *IntroductionLabel*. This set will include the *IntroductionStart* annotation and the *IntroductionEnd* annotation.

```
outputAS.add(as.firstNode(), as.lastNode(), "Introduction",
   Factory.newFeatureMap());
```

This line adds an *Introduction* annotation to the area that corresponds to this set (and thus to the *IntroductionLabel*).

```
outputAS.removeAll(as);
```

This last line removes all the annotations that are part of the set (i.e. the temporary annotations *IntroductionStart* and *IntroductionEnd*).