Homework Exercise 2

0.0.1 Quantity Scanner using JAPE

We compiled approximately 150 rules using the JAPE grammar covering the following types: mass, mileage, power, speed, density, volume, area, money, time duration, time epoch, temperature and length. The JAPE engine within GATE was used to compile and execute these rules.

JAPE is a version of CPSL[1] (Common Pattern Specification Language). JAPE provides finite state transduction over annotations based on regular expressions. The JAPE grammar requires information from two main resources: (i) a tokenizer and (ii) a gazetteer.

(1) *Tokenizer:* The tokenizer splits the text into very simple tokens such as numbers, punctuation and words of different types such as uppercase and lowercase, certain types of punctuation, *etc.* Although the tokenizer is capable of much deeper analysis than this, the aim is to limit its work to maximise efficiency, and enable greater flexibility by placing the burden on the grammar rules, which are more adaptable.

(2) *Gazetteer:* The gazetteer lists used are plain text files, with one entry per line. Each list represents a set of keywords, such as units of height, mass or speed or names of months, numeric figures in words, *etc.* An index file is used to access these lists; for each list, a major type is specified and, optionally, a minor type (though we sparingly used minor type specifications in the quantity rules). These lists are compiled into finite state machines. Any text tokens that are matched by these machines will be annotated with features specifying the major and minor types. JAPE grammar rules then specify the types to be identified in particular circumstances.

The JAPE Rule: A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The phases run sequentially and constitute a cascade of finite state transducers over annotations.

Each JAPE rule has two parts, separated by "–>". The LHS consists of an annotation pattern to be matched; the RHS describes the annotation to be assigned. A basic rule is given as:

[1] A good description of the original version of this language is in Doug Appelt's TextPro manual: `http://www.ai.sri.com/~appelt/TextPro`.

```
Rule::=
<rule> <ident> ( <priority> <integer> )?
LeftHandSide ">>>" RightHandSide
```

(1) *Left hand side:* On the LHS, the pattern is described in terms of the annotations already assigned by the tokenizer and gazetteer. The annotation pattern may contain regular expression operators (*e.g.* $*$, ?, +). There are 3 main ways in which the pattern can be specified:

1. *value:* specify a string of text, *e.g.* {Token.string == "of"}

2. *attribute:* specify the attributes (and values) of a token (or any other annotation), *e.g.* {Token.kind == number}

3. *annotation:* specify an annotation type from the gazetteer, *e.g.* {Lookup.minorType == month}

(2) *Right hand side:* The RHS consists of details of the annotations and optional features to be created. These details could also be implemented through Java code embedded within the RHS. Annotations matched on the LHS of a rule may be referred to on the RHS by means of labels that are attached to pattern elements. Finally, attributes and their corresponding values are added to the annotation. The general format of quantity extraction rules used by us is:

```
Rule: NumbersAndUnit
(({Token.kind=="number"})+:numbers
{Token.kind=="unit"})
>>>
:numbers.Name={rule="NumbersAndUnit"}
```

This says 'match sequences of numbers followed by a unit; create a *Name* annotation across the span of the numbers, and attribute rule with value *NumbersAndUnit*'.

Use of context: Context can be dealt with in the grammar rules in the following way. The pattern to be annotated is always enclosed by a set of round brackets. If preceding context is to be included in the rule, this is placed before this set of brackets. This context is described in exactly the same way as the pattern to be matched. If context following the pattern needs to be included, it is placed after the label given to the annotation. Context is used where a pattern should only be recognised if it occurs in a certain situation, but the context itself does not form part of the pattern to be annotated.

Almost all the JAPE rules for quantity annotation developed by us require Java code on the RHS, for accomplishing tasks such as associating the unit as a feature with the annotation, merging annotations, identifying two numeric quantities within a rule that identifies intervals, *etc.* Following is an example rule that identifies "Mass" quantities.

```
Rule: Mass1

(
 (
  (AMOUNT_NUMBER)+
  (({Token.string == "x"}|{Token.string == "X"}) AMOUNT_NUMBER {Token.string == "^"} AMOUNT_NUMBER)?
 ):quant1
 ({Token.string == "to"}|{Token.string == "-"})
 ({Token.string == "~"})?
 (
  (AMOUNT_NUMBER)+
  (({Token.string == "x"}|{Token.string == "X"}) AMOUNT_NUMBER {Token.string == "^"} AMOUNT_NUMBER)?
 ):quant2
 ({Token.string == "-"})?
 ({Lookup.majorType == "quantityQualifier"})?

 ({Lookup.majorType == "massunit"}):unit
)
:mass
-->
{
gate.AnnotationSet mass = (gate.AnnotationSet)bindings.get("mass");
gate.AnnotationSet unit = (gate.AnnotationSet)bindings.get("unit");
gate.AnnotationSet quant1 = (gate.AnnotationSet)bindings.get("quant1");
gate.AnnotationSet quant2 = (gate.AnnotationSet)bindings.get("quant2");

gate.FeatureMap features = Factory.newFeatureMap();
long unitBegin = unit.firstNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();
long unitEnd = unit.lastNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();
long quant1Begin = quant1.firstNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();
long quant1End = quant1.lastNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();
long quant2Begin = quant2.firstNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();
long quant2End = quant2.lastNode().getOffset().longValue() - mass.firstNode().getOffset().longValue();

features.put("unitBegin", "0"+unitBegin);
features.put("unitEnd", "0"+unitEnd);
features.put("quant1Begin", "0"+quant1Begin);
features.put("quant1End", "0"+quant1End);
features.put("quant2Begin", "0"+quant2Begin);
features.put("quant2End", "0"+quant2End);

features.put("rule", "Mass1");
outputAS.add(mass.firstNode(), mass.lastNode(), "Mass",features);
}
```