# UIMA Tokens Regex
## Documentation for developers and language experts

Damien Cram – Béatrice Daille
LINA UMR CNRS 6241
damien.cram@univ-nantes.fr
beatrice.daille@univ-nantes.fr

July 31, 2015

## Contents

UIMA Tokens Regex is an Apache UIMA Analysis Engine (AE) that implements rule-based pattern matching over a sequence of annotations, where rules are declared in a very simple regex-like language. This document aims at helping:

- the developer of a text processing flow to implement rule-based tasks easily,
- the language expert to edit the rules.

Getting UIMA Tokens Regex to work requires two mandatory phases: defining the UIMA Tokens Regex resource file (a list of rules, see section 1) and implementing the engine logic in Java (section 2).

# 1 The UIMA Tokens Regex resource file

The UIMA Tokens Regex resource file is the list of rules that UIMA Tokens Regex AE needs as input. This resource file consists in three parts, details in next subsections :

1. the file header (mandatory)
2. the definition of matchers (optional)
3. the definition of rules (mandatory)

**Example:**

```
# Header
import eu.project.ttc.types.TermSuiteTypeSystem;
use eu.project.ttc.types.WordAnnotation;
set inline = false;

# Definition of matchers
matcher A: [category=="adjective"];
matcher N: [category=="noun"];

# Definition of rules
term "an": A N;
term "nnn": N N N;
term "nn": N N;
```

## 1.1 Rule file header: import, use, set

### 1.1.1 import

The import statement is mandatory and unique. It takes as value the path to the type system definition of the CAS to analyse.

### 1.1.2 use

The use statement is mandatory and unique. It takes as value the fully qualified name of the analysed type (must be defined in the type system). The feature names available in matcher declarations (see Section 1.3) are the features contained in the type declared with the keyword use.

### 1.1.3 set

The set statement is optional and may be declared multiple times. It configures the options of the UIMA Tokens Regex's resource parser. Available options are :

| | | |
|---|---|---|
| inline | true | If set to true, matchers can be declared "inline", *ie.* directly in the right parts of the rules. When set to true, the matcher definition part is optional. |

## 1.2 Rule definition: the keyword rule

Each rule is defined with the keyword rule, a mandatory rule name, a colon ":", the regular expression part (*ie.* a sequence of annotation matchers with associated quantifiers) and a semi-colon ";".

The example below defines the rule named *My rule*, with two annotation matchers : one that matches any annotation with a feature category having a value of "noun" (occurring 1 to $n$ times sequentially), followed by one that matches exactly one annotation with a feature category having a value of "adjective".

```
rule "My rule": [category=="noun"]+ [category=="
    adjective"];
```

## 1.3 Matcher definition

There are two types of matcher, each defined with its own syntax : the The feature structure matcher and the covered text string matcher.

### 1.3.1 The feature structure matcher, defined with [ ]

The feature structure matcher applies to Apache UIMA feature structures, *ie.* annotations. It is a boolean expression defined within brackets [ ].

Each boolean expression has the form feature op literal. In the example below, the rule has one matcher, where feature is *category*, op is the equality operator =, =, and literal is the string value *"noun"*.

```
rule "My rule": [category=="noun"];
```

Boolean expressions within brackets can also be composed of sub-expressions with the help of parentheses and three types of boolean operators :

| operator | description |
|---|---|
| & | the logical conjunction |
| \| | the logical disjunction |
| ! | the logical negation (not implemented yet) |

The operators & and | are n-ary:

```
rule "My rule": [category=="noun" & (lemma == "chair
   " | (stem=="mang" & mood="participle" & tense="
   past") | lemma=="vitesse")];
```

**Features**

All features declared in the *used type* (see Section 1.1.2) can be referenced from the left part of a boolean expression in matchers. For example, if the used type is `WordAnnotation` as defined on figure 1, the available features would be:

| feature | literal type |
|---|---|
| `begin` | `Integer` |
| `end` | `Integer` |
| `category` | `String` |
| `lemma` | `String` |
| `stem` | `String` |
| `tag` | `String` |
| `subCategory` | `String` |
| `number` | `String` |
| `gender` | `String` |
| `case` | `String` |
| `mood` | `String` |
| `tense` | `String` |
| `person` | `String` |
| `possessor` | `String` |
| `degree` | `String` |
| `formation` | `String` |

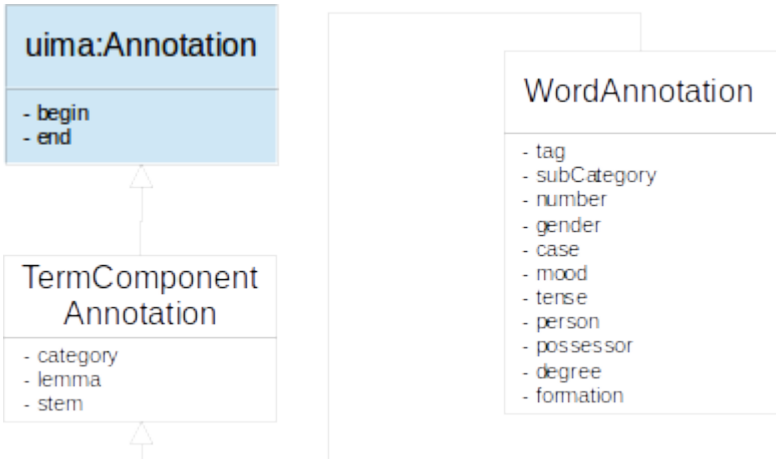



Figure 1: Example used type: `WordAnnotation`

**Operators**
There exists 6 different operators :

| operator | description |
|---|---|
| == | the equality |
| != | the inequality |
| <= | less than or equal to |
| < | less than |
| >= | greater than or equal to |
| > | greater than |

**Literal types**
Four different types of literals are defined:

| Literal type | Possible values in `UIMA Tokens Regex` syntax | Mapped `Apache UIMA` types |
|---|---|---|
| `String` | "noun", "chair", "axe" | `String` |
| `Integer` | 123 | `Integer`, `Short`, `Long` |
| `Float` | 123.456 | `Float`, `Double` |
| `Boolean` | true, false | `Boolean` |

`Apache UIMA` array and list types and associated collection operators (`in`, `not in`, etc.) are not yet supported by `UIMA Tokens Regex`.

### 1.3.2 The covered text string matcher, defined with / /

The covered text string matcher applies to any `Apache UIMA` textual annotation, *ie.* annotation with `begin` and `end` features applying on string SOFAs. It is a `Java` regular expressions language defined between two slashes "/" that matches against the raw text covered by the annotation, *ie.* the value returned by the method `getCoveredText()`.

For example, the rule below has exactly one matcher that matches any annotation covering a text starting with "mang".

```
rule "My rule": /^mang.*$/;
```

## 1.4 Matcher predefinition: the keyword `matcher`

Matcher predefinitions are a list a `matcher` statements occurring after the header and before the `rule` statements.

So far, our examples showed matchers defined *inline*, *ie.* directly on the right part of the rule. The keyword `matcher` allow to predefine matchers before they are used in rule declaration.

```
matcher M1 : /^mang.*$/;
matcher M2 : [category=="noun"];

rule "My rule": M1 M2;
```

The matcher predefinition declares a *matcher label* (`M1` and `M2` in the example above) and the actual matcher definition.

Matcher labels can be invoked both in rule statements and in other matcher statements. Pay attention that a matcher label invoked in a matcher statement must be defined prior to this statement.

```
matcher B : /^mang.*$/;
matcher B2 : [B & category=="noun"];

rule "My rule": B2+;
```

With such a predefinition syntax, a matcher can be reused several times in different rules, without having to copy many times the same matcher code, hence improving the readability of the rule list.

## 1.5 The keyword `as`

The keyword `as` can be optionally used in a matcher predefinition to rename its matcher label.

```
matcher A1 as A: [category=="adjective"];
matcher A2 as A: [category=="verb" & tense="past" &
   mood="participle"];
```

```
matcher N: [category=="noun"];

rule "My rule 1": A1 N;
rule "My rule 2": A2 N;
```

For both rules above, the sequence of matcher labels attached to each rule occurrence extracted by `UIMA Tokens Regex`'s engine (see Section 2) will be `A N`, instead of `A1 N` for the first rule and `A2 N` for the second rule, because of the `as` keyword.

The keyword `as` has no effect in rule declaration nor in the way the `UIMA Tokens Regex` engine extracts rule occurrences, it only renames labels for later reuse by other AEs in the `Apache UIMA` AE process flow.

### 1.6 The quantifiers: `?`, `*`, `+`, `n`, `m,n`

Any matcher can be associated with one the the following regular expression quantifiers.

| Quantifier | Quantity of successive matching annotations |
|---|---|
| `?` | zero or one |
| `*` | any number (zero or $n$) |
| `+` | at least one (one or $n$) |
| `n` | exactly $n$ |
| `?` | from $m$ to $n$ ($m \leq n$) |

Example:

```
matcher A: [category=="adjective"];
matcher N: [category=="noun"];
matcher N: [category=="preposition"];

# Matches NA, NRA, NRRA, NRRRA, etc.
rule "My rule 1": N R* A;

# Matches NPN and NPDN
rule "My rule 2": N P D? N;

# Matches NA, NAA, NAAA, etc.
rule "My rule 3": N A+;
```

The quantifiers can be associated to any type of matcher : feature structure matchers, regular expression matchers, and matcher labels :

```
matcher N: [category=="noun"];
matcher N: [category=="adjective"];
```

```
# A quantifier associated to a feature structure
   matcher
rule "My rule 1": N [category="adverb"]* A;

# A quantifier associated to a regular expression
   matcher
rule "My rule 2": N P /^(de)|(des)|(du)$/? N;

# A quantifier associated to a matcher label (
   predefined matcher)
rule "My rule 3": N A+;
```

### 1.7 The any matcher []

The any matcher `[]` matches any annotation. It is the equivalent of the dot "`.`" in string regular expressions.

```
# Matches any term of one of the forms: NPN, NPDN,
   NPAN, NPDAN, etc.
rule "complex term 1": N P []* N;
```

### 1.8 Ignoring labels with ∼

In the right part of rule declaration, any matcher label can be prefixed with the special character ∼ to refrain the following label to appear in the rule occurrences extracted by `UIMA Tokens Regex`'s engine.

Each time the engine finds a matching occurrence for a rule, it produces an instance of the class `Occurrence` (see Section 2). The class `Occurrence` has a method named `getPattern()` returning the sequence of matcher labels that have matched. The effect of the special character ∼ is to remove some labels from the pattern.

This functionality is useful when some annotation are mandatory to specify a matching context but useless for further analysis.

The example illustrates this functionality in the case of terminology extraction. We need the rule `My rule` to extract all term having the pattern `NPN`, but we need to appear after a determiner `D`. As we don't want the `D` to appear in the term's pattern, we hide it from the pattern by prefixing it with ∼.

```
matcher N: [category=="noun"];
matcher P: [category=="preposition"];
matcher D: [category=="determiner"];

# The following rule matches DNPDN or DNPN
```

```
# but getPattern() will always return NPN
rule "My rule": ~D N P ~D? N;
```

The character ~ does not apply to feature structure matchers nor to regular expression matchers.

### 1.9 Commenting with #

Any sequence of characters starting with # will be considered as a comment and will be ignored by the UIMA Tokens Regex parser until the end of the line.

```
# This is ignored
rule "My rule 1": [category=="noun"]

rule "My rule 2": [category=="adjective"];# This is
   ignored
```

### 1.10 Example rule list resource file

The example below is a complete example of a UIMA Tokens Regex rule list that is used in the context of complex term detection over french word annotations.

```
import eu.project.ttc.types.TermSuiteTypeSystem;
use eu.project.ttc.types.WordAnnotation;
set inline = false;


matcher N: [category=="noun"];
matcher Nn as N: [N | category=="name"];
matcher Nnn as N: [Nn | category=="numeral"];
matcher R: [category=="adverb"];
matcher P: [category=="adposition" & subCategory="
   preposition"];
matcher Pde as P: [P & lemma=="de"];
matcher V: [category=="verb"];
matcher A: [category=="adjective"];
matcher Vpp as A: [V & mood=="participle" & tense="
   past"];
matcher A2 as A: [A | Vpp];
matcher A3 as A: [A | category=="numeral"];
matcher C as C: /^(et|ou)$/ ;
matcher D as D: [category=="determiner" | category==
   "article"];
```

```
matcher L1: [lemma=="axe" | lemma=="calage" | lemma
   =="chair" | lemma=="couleur" | lemma=="dé
   veloppement" | lemma=="état" | lemma=="face" |
   lemma=="genre" | lemma=="origine" | lemma=="pas"
   | lemma=="pâte" | lemma=="phase" | lemma=="type"
   | lemma=="vitesse" | lemma=="voie" ];
matcher L2: [lemma=="cage" | lemma=="effacement" |
   lemma=="effet" | lemma=="fonction" | lemma=="voie
   "];
matcher N1 as N: [N & L1];
matcher N2 as N: [N & L2];
matcher comma: /,/;
matcher Og: /"|≪/;
matcher Fg: /"|≫/;

rule "nn": N Nn;
rule "na": N A2 ;
rule "nra": N R+ A;
rule "naa": N A A;
rule "npn": N P ~D? N ;
rule "nnpn": N N P ~D? N;
rule "npnn": N P N1 Nnn; # Never matches
rule "npnqnq": N P N1 ~Og Nnn ~Fg;
rule "naca": N A C A;
rule "naca+": N A comma A C A;
rule "npna": N P N A;
rule "npan": N P A3 N;
rule "napn1": N A2 P N;
rule "napan":  N A P A3  N;
rule "dncdna": ~D N C ~D N A;
rule "dncdnpn": ~D N C ~D N P N ;
rule "npncpn": N P N C P N;
rule "npdncpdn": N P ~D N C P ~D N;
rule "npncpn": N P N C P N;
rule "npn,pncpn": N P N comma P N C P N;
rule "npnpn": N P N2 Pde Nn;
```

## 2 UIMA Tokens Regex Analysis Engine (AE)

The UIMA Tokens Regex AE takes its resource file (*cf.* Section 1) as input, scans all annotation sequentially (see Section 2.1 for more details) and notifies a Java handler method, which must be implemented (See Section 2.2), every time a rule matches.

## 2.1 Sequential scanning of annotations

The `UIMA Tokens Regex` AE parses the rule list and creates for each rule exactly one finite state automaton, where each annotation matcher plays the role of a transition label (There are as many annotation matchers as there are transitions). The engine iterates over the sequence of all annotations of the type declared by the keyword `set` (*cf.* Section 1.1.3) taken in their natural `Apache UIMA` order (`begin` ascending, `end` descending). The engine propagates the current annotation to all automaton, *ie.* to $n$ automata, where $n$ is the number of rules declared.

Each time an automaton fails, *ie.* an automaton cannot propagate the current annotation, the AE behaves as follows:

if the automaton was in an acceptable state before failing, then a new occurrence of the rule is created and the handler method `ruleMatched()` is invoked. All other automaton instances of the same automaton are dropped,

if the automaton was in no acceptable state before the annotation, the AE drops the current automaton instance and tries to create a new one with the current annotation.

We can deduce two important consequences of this behaviour:

1. the AE will never produce two overlapping occurrences of a single rule,
2. occurrences of different rules are produced independently from each other can overlap.

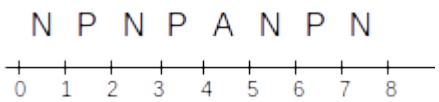
**Example:**

Let's consider the two following rules :

```
matcher N : [myFeat="N"];
matcher P : [myFeat="P"];
matcher A : [myFeat="A"];

rule "Rule1": N P N;
rule "Rule2": N P A N;
```

And the following sequence of annotations (actually figure below represents only the values of the annotations for the feature `myFeat`):




The occurrences recognized by the AE will be :

| | |
|---|---|
| `Rule1`'s occurrence: | $[0, 3]$, $[5, 8]$, $[7, 10]$ |
| `Rule1`'s occurrence extracted by the AE: | $[0, 3]$, $[5, 8]$ |
| `Rule2`'s occurrence: | $[2, 6]$ |
| `Rule2`'s occurrence extracted by the AE: | $[2, 6]$ |
| **Final occurrences set:** | `Rule1`:$[0, 3]$, $[5, 8]$, `Rule2`:$[2, 6]$ |

In this example, the occurrence $[5, 8]$ of `Rule1` overlaps with the previous occurrence of the same rule $[0, 3]$. It is then not recognized by the AE.

## 2.2 Definition of the `Java` handler method

As explained in Section 2.1, every time an occurrence is recognized, the method `ruleMatched()` is invoked. All the developer has to do is to extends the class `TokenRegexAE` and to override its methods (including `ruleMatched()`) so as to implement the desired behaviour of the AE.

### 2.2.1 Extending the class `TokenRegexAE`

Here is the list of the abstract methods of the class `TokenRegexAE` that the user can override :

| Method | Description |
|---|---|
| `ruleMatched(JCas, Occurrence)` | Invoked whenever a rule has matched the sequence of annotations. It is given the `Apache UIMA` CAS object and an instance of the class `Occurrence` with all the details about the matching rule and the matched annotation. (See Section 2.2.2) |
| `beforeRuleProcessing(JCas)` | Invoked before the AE starts processing the sequence of annotations. This is where the developer can implement pre-processing instructions. |
| `afterRuleProcessing(JCas)` | Invoked after the AE ends processing the sequence of annotations. This is where the developer can implement post-processing instructions. |

### 2.2.2 The `Occurrence` class

Whenever a rule matches in the sequence of annotations, the AE invokes the method `ruleMatched()` and passes to it an instance of the class `Occurrence`. Here is the list of public methods of this class that the developer can invoke so as to access all matching information available.

| Method | Description |
|---|---|
| `int getBegin()` | returns the begin offset of the matching occurrence in the SOFA string. |
| `int getEnd()` | returns the end offset of the matching occurrence in the SOFA string. |
| `Rule getRule()` | returns the rule that this occurrence has matched, as a `Rule` object (see Section 2.2.3). |
| `List getAllMatchingAnnotations()` | returns the list of all annotations that make the occurrence of the matching rule. The list includes ignored annotations, *ie.* annotations being prefixed with $\sim$ (see Section 1.8). |
| `List getLabelledAnnotations()` | returns the list of annotations that make the occurrence of the matching rule (without ignored annotations) |
| `List getLabels()` | returns the list of the labels of the annotations that make the matching occurrence. |
| `String asPatternString()` | returns the string representation of the labels returned by `getLabels()`, *ie.* the labels joined with white spaces. |
| `int size()` | returns the number of annotations that make the occurrence of the matching rule, *ie.* the size of the list returned by the method `getLabelledAnnotations()`. |

### 2.2.3 The `Rule` class

Each occurrence object has a reference to its matching rule. This rule is an instance of the class `Rule` and has the following public methods.

| Method | Description |
|---|---|
| `String getName()` | Returns the rule name as declared by in the left part of the rule statement. |

## 2.3 Example implementation of `UIMA Tokens Regex` AE

In the example implementation below, we create on each occurrence recognized an annotation of type `MultiWordTermOccurrence` and we set its `begin`, `end`,`pattern`, and `category` features from the `Occurrence` object and and its associated rule's name and id.

```
public class RegexSpotter extends TokenRegexAE {
  @Override
  public void ruleMatched(JCas jCas, Occurrence occ) {
    Annotation annotation = jCas
        .getCas().createAnnotation(
            jCas.getCasType(MultiWordTermOccurrence.type),
            occ.getBegin(),
            occ.getEnd());

    MultiWordTermOccurrence a =
      (MultiWordTermOccurrence) annotation;

    StringArray patternFeature =
      new StringArray(jCas, occ.size());
    i = 0;
    for(String l:occ.getLabels()) {
      patternFeature.set(i, l);
      i+=1;
    }

    a.setPattern(patternFeature);
    a.setCategory(occ.getRule().getName());

    a.addToIndexes();
  }
}
```