GATE JAPE Grammar Tutorial

Version 1.0

Dhaval Thakker, PA Photos, UK
Taha Osman, Nottingham Trent University, UK
Phil Lakin, PA Photos, UK

February 27, 2009

Table of Contents

Table of Contents 2
Introduction to General Architecture of Text Engineering (GATE) 3
JAPE Rules 6
Example 1. A simple example to decide the category of sports 7
Example 2. Multiple patterns in JAPE grammar 10
Example 3. Nested Patterns 11
Example 4. Using Part of Speech (POS) features to extract entities 15
Example 5. Priority in JAPE rules 18
Example 6. Handling repetitiveness in patterns using Macro 23
Example 7. Using negation operator in JAPE 24
Example 8. Using JAVA in RHS of JAPE Grammar 25
Example 9. Using a common file as a holder of application specific JAPE grammar files 26
Example 10. Using JAVA in RHS of JAPE: A complex example 27
Example 11. Using Split to control the application of a rule to a single sentence. 32
Example 12. Co referencing 33
Example 13. Creating Temporary annotations and then deleting at the end when it is no longer useful. 36
Example 14. Creating new entities to use in the JAPE grammar 36
Bibliography 38

Conventions used in this tutorial:

MEANING	Meaning of a term
TIP	Tips on how to do things
NOTE	Worth Remembering or referring back

For improving the readability of the text, important terms are written in *times new roman, italic,* and the examples are written in `Courier New, italic` font such as in the following:

```
Phase: firstpass                                    //1
Input:  Lookup                                      //2
```

The tutorial includes number of examples that are referred in the text here. The examples are in the tutorial folder.

In order to follow this tutorial you must have GATE 5.0 or later version, which is available from the GATE website.

Introduction to General Architecture of Text Engineering (GATE)

The writer of this tutorial assumes basic understanding of the GATE system and the Java programming language. To be more specific, readers shall be familiar with the working of the GATE interface and ANNIE (A Nearly-New IE system) processing resources of sentence splitter, tokeniser, POS Tagger, gazetteer and JAPE transducer. A brief introduction is given below to these components with respect to their use in this tutorial.

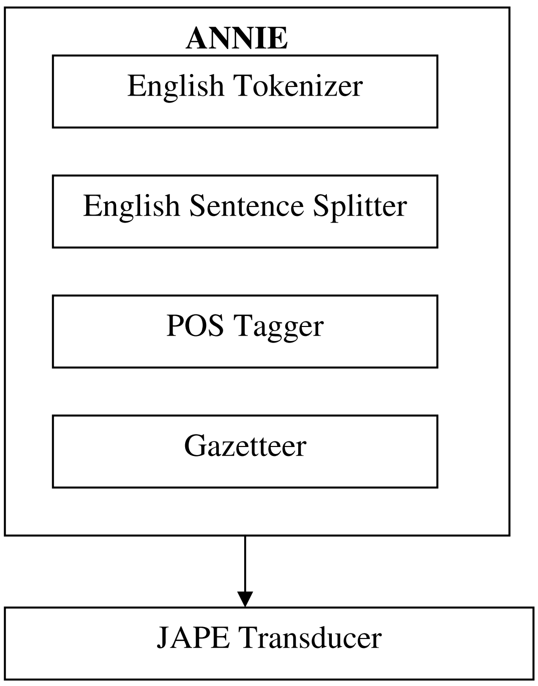


Figure 1 Typical GATE system components

The *tokeniser* splits the text into very simple tokens such as numbers, punctuation and words of different types. For example, GATE distinguishes between words in uppercase and lowercase, and between certain types of punctuation. The aim is to limit the work of the tokeniser to maximise efficiency, and enable greater flexibility by placing the burden on the grammar rules, which are more adaptable.

The *sentence splitter* is a cascade of finite-state transducers which segments the text into sentences. This module is required for the POS tagger and other modules.

MEANING

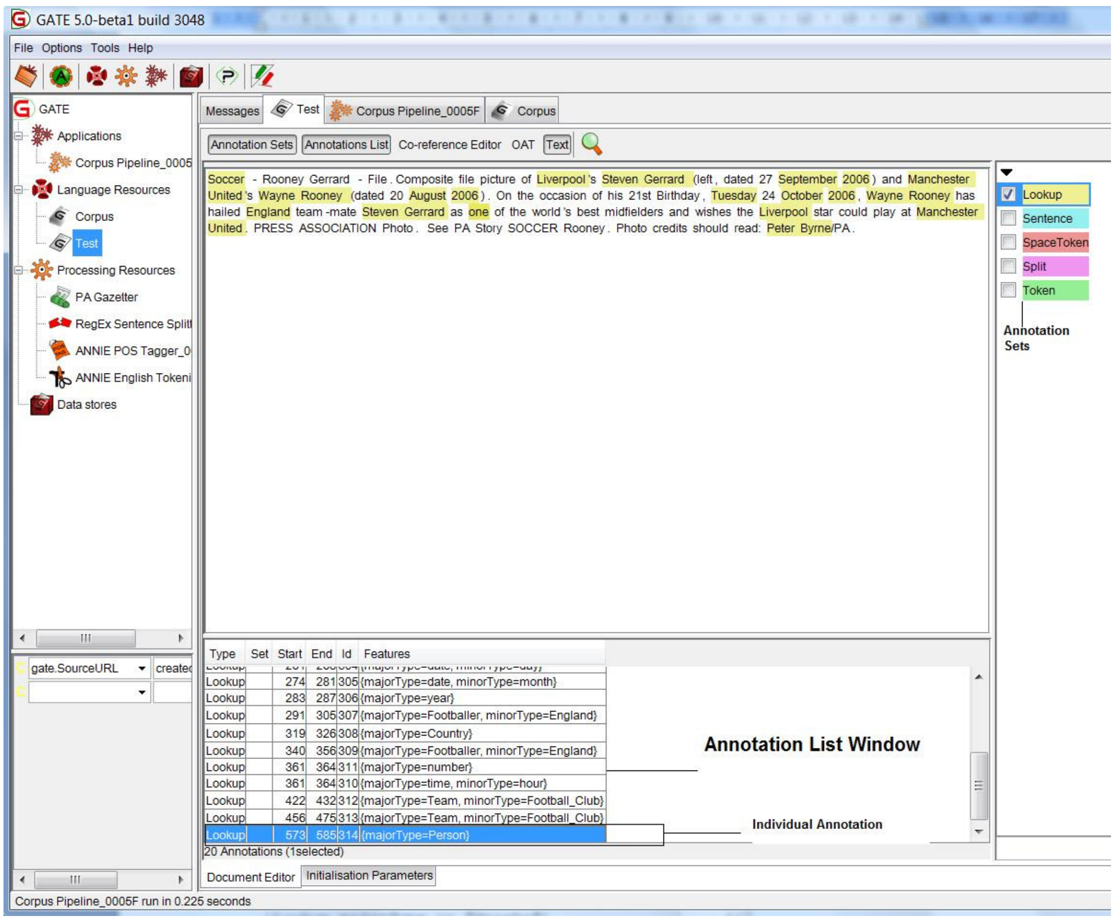
In general terms a transducer is a device that converts one type of energy to another for various purposes including measurement or information transfer.

In the context of GATE system here and NLP in general, transducer is a finite state transducer with two tapes: an input tape and an output tape. On this view, a transducer is said to transduce (i.e., translate) the contents of its input tape to its output tape, by accepting a string on its input tape and generating another string on its output tape.

The tagger produces a part-of-speech tag as an annotation on each word or symbol. The list of tags used is given in Appendix E of GATE user guide [1].

The gazetteer list is the lookup list of entities. They are stored in various files which the GATE Gazetteer uses to detect in generally the initial phases of annotations. JAPE rules utilize these annotations along with annotations from other processing resources to further identify patterns/entities from the text.

The GATE framework provides the ability to cascade or chain individual JAPE transducers together one after another. This chaining allows later JAPE transducers to work upon the output of earlier JAPE transducers thereby building more and more complex annotations and incorporating more of the context (semantics) of the document into the new annotations [2].



Soccer - Rooney Gerrard - File . Composite file picture of Liverpool 's Steven Gerrard (left , dated 27 September 2006) and Manchester United 's Wayne Rooney (dated 20 August 2006) . On the occasion of his 21st Birthday , Tuesday 24 October 2006 , Wayne Rooney has hailed England team -mate Steven Gerrard as one of the world 's best midfielders and wishes the Liverpool star could play at Manchester United . PRESS ASSOCIATION Photo . See PA Story SOCCER Rooney . Photo credits should read: Peter Byrne/PA .

Type	Set	Start	End	Id	Features
Lookup		274	281	305	{majorType=date, minorType=month}
Lookup		283	287	306	{majorType=year}
Lookup		291	305	307	{majorType=Footballer, minorType=England}
Lookup		319	326	308	{majorType=Country}
Lookup		340	356	309	{majorType=Footballer, minorType=England}
Lookup		361	364	311	{majorType=number}
Lookup		361	364	310	{majorType=time, minorType=hour}
Lookup		422	432	312	{majorType=Team, minorType=Football_Club}
Lookup		456	475	313	{majorType=Team, minorType=Football_Club}
Lookup		573	585	314	{majorType=Person}

20 Annotations (1selected)

Corpus Pipeline_0005F run in 0.225 seconds

Figure 2 GATE GUI and various options (Annotation List and Sets, Individual Annotation)

Annotation is the act of tagging, commenting or marking media with some metadata information to identify type or the content of the media.

In the context of GATE annotation it means creating new or identifying existing metadata information. For example, identifying person name from the text.

We will concentrate on JAPE grammar in this tutorial. Please refer to the GATE user guide for further information on other resources. The terms we refer often in this tutorial are *annotation lists*, *annotation sets* and *individual annotations* are highlighted in the Figure 2.

Let us make JAPE a **J**olly **A**nd **P**leasant **E**xercise for you.

Important: *This tutorial uses two gazetteer lists for running examples. To avoid the confusion, already annotated text files are stored in a data store. Hence to run an example JAPE grammar you shall skip the tokeniser, splitter and tagger phases and just have to run the JAPE transducer. You will have to load the data store in the GUI before running transducer.*

JAPE Rules

A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. The grammar always has two sides: Left and Right. The LHS of the rule contains the identified annotation pattern that may contain regular expression operators (e.g. *, ?, +). The RHS outlines the action to be taken on the detected pattern and consists of annotation manipulation statements. Annotations matched on the LHS of a rule are referred on the RHS by means of labels. For example,

LHS (regular expression for annotation pattern):

i.e., Lookup for annotation player and label it *player*

RHS (manipulation of the annotation patter from LHS):

i.e., Get the gender of the *player*, if gender=male re-label *player* as *Male-Player,* else re-label as *Female-Player.*

To explain further, we will take a simple example:

Example 1. A simple example to decide the category of sports

Load *Example 1.txt* document in the GATE GUI from the data store. In the GUI annotations list window, you will be able to see number of annotations. So far we have not used any JAPE rule and all the annotations are managed through the lookup lists from the Gazetteer.

Notice that the type of default annotation is *"Lookup"*. Now let's write our first JAPE rule as we don't like everything under *Lookup* and would like annotating entities according to their right labels. For example, first entity being detected is *Soccer {Type = Lookup, start=0, End=6, id=295, majorType=Sports}* and we want to label *Soccer* as *"Sports"*. The rule to achieve this is below:

```
Phase:firstpass                                 //1
Input:  Lookup                                  //2
Options: control = brill                        //3

Rule: SportsCategory                            //4
Priority: 20                                    //5
(                                               //6
{Lookup.majorType == "Sports"}                  //7
): label                                        //8
-->                                             //9
:label.Sport = {rule= "SportsCategory" }        //10
```

JAPE Grammar 1 SportsCategory.jape

NOTE

```
There is no standard JAPE editor available; however from the
available editors Vim (an improved version of UNIX vi editor;
available for windows) does a decent job of highlighting things in
colour and maintaining the brackets.  Download vim, install it. To
make JAPE files treated as Java files, go to the filetype.vim from
the installation directory and change the following line

from:
" Java
au BufNewFile,BufRead *.java,*.jav  setf java
to:
" Java
au BufNewFile,BufRead *.java,*.jav,*.jape setf java

For associating Jape files to Java editor in Eclipse, you have to
associate ".jape" with the Java editor.  In Eclipse on the
Microsoft Windows, you go to the menu at Window | Preferences |
General | Editors | File Associations and then you associate the
default Java editor with *.jape. (This is not verified by this
author)
```

Explanation of the rule:

Line 1: A JAPE grammar consists of a set of phases, each of which consists of a set of pattern/action rules. These phases have to be uniquely labelled; here we label our annotation phase as *firstpass*. Do note that the name of the phase can be different from the name of the jape grammar file.

Line2: Input annotations must also be defined at the start of each grammar, these are the annotations against which the rule will be matched. If no annotations are defined, the default will be *Token*, *SpaceToken* and *Lookup* hence by default only these annotations will be considered when attempting a match. Every annotation type that is going to be matched in that grammar phase must be included in the *Input* set. Any annotation type that is not defined will be ignored in the pattern matching.

TIP

Always remember to include input annotations that you need to use in JAPE grammar rule. By default the transducer will include *Token*, *SpaceToken* and *Lookup*.

Line3: At the beginning of each grammar, several options can be set. They could be:

Control - this defines the method of rule matching. Possible options are {Appelt, Brill, All, Once}. More details on these methods are given in the Example 5.
Debug - when set to true, if the grammar is running in *Appelt* mode *(control option)* and there is more than one possible match, the conflicts will be displayed in the messages window.

Lin4: Name a rule, in this example the name of the rule is *SportsCategory.*

Line5: Explained later in the Example 5. For the time being assume that the priority field *(Priority: X)* are used for setting priority of the rule over other rules in the same grammar file.

Lines 6,7 and 8:

(

```
        {Lookup.majorType == "Sports"}
): label
```

This is important part of the rule. What we are saying to transducer with this rule is this: find out annotations with the pattern *Lookup.majortype* == *"Sports"* and temporarily name it *"label"*. Syntax-wise == and { } are important. The patterns you are looking for has to be surrounded with { } and under the same closing (). This will become clearer with more examples.

Line 9 `-->` is the boundary of the LHS rule. What ever follows this will be RHS Of the rule!

Line 10

```
:label.Sport = {rule= "SportsCategory"}
```

Here we are saying to the transducer that temporary label (from line 8) will be renamed to *"Sport"* and the rule that achieves this is *"SportsCategory"*. Naming a rule here is important for the debugging purpose as when the rule fires, it will be part of the annotation properties that you can see in GATE GUI. In this example you will see *Soccer* annotated as *{ rule=SportsCategory}.*

The syntax of the line 10 is important to note,

```
temporary label. New label = {rule= "name of the rule"}
```

This is a simple example, hence we are not doing much with the RHS of the rule, but as we go further you will see that we can detect complex patterns from LHS rule and can do useful stuff on the RHS. Even we can use Java in the RHS….horrayyyyyy!!

Exercise 1: Write your first JAPE Grammar

Write a JAPE grammar for the text in the *Example 1.txt* file that will create an annotation for *"Footballer"*.

Hint: You have to use the existing setup for the example 1 which already has a *Lookup* annotation with *Footballer* as *majorType*.

Example 2. Multiple patterns in JAPE grammar

It is possible to have more than one pattern and corresponding action in a single JAPE grammar rule, as shown in the *TwoPatterns.jape* rule below. On the LHS, each pattern is enclosed in a set of round brackets and has a unique label; on the RHS, each label is associated with an action. In this example, the Lookup annotation *Title* is labelled “title” and is given the new annotation *Title*; the *Person* annotation is labelled “person” and is given the new annotation *Person*.

```
Phase:firstpass
Input:  Lookup Token
Options: control = brill

Rule: TwoPatterns
Priority: 20
(
{Lookup.majorType == "Title"}
): title
(
{Lookup.majorType == "Person"}
):person
-->
:title.Title ={rule= "TwoPatterns" },
:person.Person = {rule= "TwoPatterns" }
```

JAPE Grammar 2 TwoPatterns.jape

Load this grammar as a JAPE transducer and run on the *example2.txt* file from the data store.

Notice in the annotation list that the *Title (=Sir)* has been selected twice. This happens because the *Person* is annotated as *“Alex”* and *“Alex Ferguson”* (as it happens to be in the Gazetteer lookup). Hence the rule goes through the loop twice and outputs both without bothering about the longest match.

Let’s try a variation. First of all delete the annotations marked with “Person” and “Title”. Now, change the control option to “appelt” and see how the transducer behaves. The modified rule is already available in the example 2 folder and is called *“TwoPatternsWithAppelt.jape”*.

As you have rightly guessed the results have something to do with what you specify in the control field. Remember from example 1, the options for the control field are “brill”,” appelt”, “all”, “first” and “once”.

The options for the control field are "brill"," appelt", "all", "first" and "once".

Among these, apart from "brill" you will be using "appelt" the most often. We will explain these options in eExample 5.

Example 3. Nested Patterns

Let's first define the problem. Here is the text story (Example 3.txt),

AC Milan player David Beckham is going to comment on his future with LA Galaxy, who are eager to keep him in USA.

We want to create a grammar rule to detect the name of the player based on the pattern

If mention of the word "player" followed by a name of a person

Then the person = a player.

Following is the rule that achieves this:

```
Phase:nestedpatternphase
Input:  Lookup Token
//note that we are using Lookup and Token both inside our rules.
Options: control = brill
Rule: playerid
(
 {Token.string == "player"}
)
:temp
(
{Lookup.majorType == Person}
|
(
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
)

)
:player
-->
 :player.Player= {rule = "playerid"}
```

JAPE Grammar 3 nestedpatten.jape

We separate the pattern into two, where the first one is:

```
(
 {Token.string == "player"}
): temp
```

{Token.string == "player"} captures the first part of the pattern. Note that if you want to imply that word “player” can be mentioned as “Player”, you could replace it with {Token.string =~ "[Pp]layer"}. Note the equality signs changes from == to =~.

There are 3 main ways in which the pattern can be specified: following.

- specify a string of text, e.g. {Token.string == "of"}
- specify the attributes (and values) of an annotation. Several operators are supported (Equality operators (“==” and “!=”))
- Comparison operators (“<”, “<=”, “>=” and “>”) and Regular expression operators (“=~”, “==~”, “!~” and “!=~”)
    - {Token.kind == "number"}, {Token.length != 4} - equality and inequality.
    - {Token.string > "aardvark"}, {Token.length < 10} - comparison operators. >= and <= are also supported.
    - {Token.string =~ "[Dd]ogs"}, {Token.string !~ "(?i)hello"} - regular expression. ==~ and !=~ are also provided, for whole-string matching.
    - {X contains Y} and {X within Y} for checking annotations within the context of other annotations.

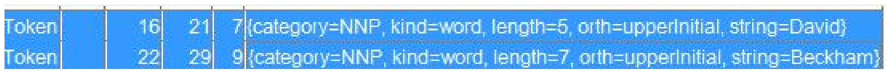
We will cover some of these regular expressions in our examples.

Coming back to the example:

The next pattern is:

```
(
{Lookup.majorType == Person}
|
(
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
)

)
```

Here we are trying to use the existing annotation *(Person)* through Lookup and as an option where the *Lookup* type *Person* is not available we are trying to using the Part of Speech (POS) annotation where kind=word, category=NNP and orth=upperInitial. This when repeated will give us a good indication of the fact that the pattern could be *firstname* and *lastname* of a person. The identification of the POS patterns you need to use could be debugged in the GUI before creating a general rule. For example, you know that *firstname* and *lastname* of a person will be a *Noun*, and you can check on the annotation list to identify how the POS tagger tags such nouns. Here is a snapshot of such annotations,



```
Token      16   21   7 {category=NNP, kind=word, length=5, orth=upperInitial, string=David}
Token      22   29   9 {category=NNP, kind=word, length=7, orth=upperInitial, string=Beckham}
```

The list of tags used is given in Appendix E of GATE user guide [1].

Load the *Example 3.txt* and run the JAPE transducer over the text. Check the results.

Consider what will happen if there was another word between word “player” and name of the person? To see the effect, try another example (Example 3a) from the data store. It will not yield any results. This is because above pattern expects token *player* immediately preceded by a person name, which is not the case with this example. Following alternative rule *(nestedpattern-alt.jape)* will work.

```
Phase:nestedpatternphase
Input:  Lookup Token
//note that we are using Lookup and Token both inside our rules.
Options: control = appelt


Rule: playerid
(
 {Token.string == "player"}
 {Token}
)
:temp
({Lookup.majorType == Person}
 |
 (
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
 )
)
:player
-->
 :player.Player= {rule = "playerid"}
```

JAPE Grammar 4 nestedpattern-alt.jape

Hence when you are sure that there are going to be an unknown token you can keep it abstract as shown in the above example. But what if you are not too sure?

You can use ({Token})? Which uses? operator to address this problem (see *nestedpattern-alt2.jape*).

```
Phase:nestedpatternphase
Input:  Lookup Token
//note that we are using Lookup and Token both inside our rules.
Options: control = appelt


Rule: playerid
(
 {Token.string == "player"}
 ({Token})?
)
:temp
({Lookup.majorType == Person}
|
 (
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
 )
)
:player
-->
 :player.Player= {rule = "playerid"}
```

JAPE Grammar 5 nestedpattern-alt2.jape

The grammar *nestedpattern-alt2.jape* will work with both examples ex3 and ex3a. Discuss why?

Next stage,

How about applying nestedpattern-alt2.jape for the following example text*(Example 3b.txt)*:

"AC Milan player model mid fielder and englishman David Beckham is going to comment on his future with LA Galaxy, who are eager to keep him in USA."

The option we can use to address the problem is "*" to (see *nestedpattern-alt3.jape*). The problem in using such regular expression is the obvious performance penalty. Hence it is important that we have a good guess of the language pattern in our text to write these rules.

```
Avoid the use of "*" based regular expressions to
improve the performance of the transducer.
```

Exercise 3:

Let's look at a scenario that only uses tokens to detect a pattern from the text. For example, IP address is the combination of pattern (number.number.number.number) and you can write a JAPE Grammar that uses POS Token to detect this pattern. As an exercise write and run such grammar to detect *IPAddress* entity from the following text (*exercise3.txt* from data store)*:*

```
"The role internet has played in our lives is quite remarkable. Who
knew we will be known by some numbers such as 132.123.1.22 ?"
```

For answer see the *exercise.jape* file.

Example 4. Using Part of Speech (POS) features to extract entities

Let's say that we want to identify a sports location through its context. For example, identifying a stadium location where the word *"Stadium"* is used in various combinations, i.e. *"Emirates Stadium", "Wembley Stadium*", and *"Ben Hill Griffin Stadium"*. We will have to consider these generalities to write a generic rule that is applicable in all these contexts. Let's target an example text story (fictional) for writing this rule (Example 4.txt):

"The sound of the Millennium Stadium when a crowd is watching a football or rugby match is amazing. If you sit right at the back of a full stadium, you can experience a tidal wave of cheering approach you from the other side. Invisible, but just as infectious.

Testing other examples of Sardar Patel Stadium , Emirates Stadium and Wembley Stadium and Ben Hill Griffin Stadium . However this shall not be picked up Stadium."

True positives we are after:

Millennium Stadium
Sardar Patel Stadium
Emirates Stadium
Wembley Stadium
Ben Hill Griffin Stadium

False positive we want to ignore are:

Stadium

We can use POS features to achieve the desired results. We know that word *"Stadium"* can be used in conjunction with one or two words *(Emirates or Ben Hill Griffin)* that must be in upper initial letters, is represented as Token *word (Token.kind== "word")* and is a Noun hence *Token.category=NNP (Noun singular)* or *NNPS (Noun Plural).*

Looking at the POS output in the GUI annotation list and selecting Token from the list of annotations, will give you lots of information about this. You can study number of examples in this manner and can generalize them. The jape grammar that achieves required objectives is *locationcontext1.jape* from the Example 4 folder and the text story used for testing is the Example 4.txt from the data store.

```
(//1
(//2

        (3
           {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
            |
         {Token.kind == word, Token.category == NNPS, Token.orth == upperInitial}
        )//3
        (//4
         {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
          |
         {Token.kind == word, Token.category == NNPS, Token.orth == upperInitial}
        )?
 ) //2

(5
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
   |
  {Token.kind == word, Token.category == NNPS, Token.orth == upperInitial}
)?  //5
)//1
```

JAPE Grammar 6 locationcontext1.jape

Let’s understand how this is achieved. To simplify how looping works, the grammar is coloured coded in above. Loop 1 is the whole pattern we are trying to match and consist of pattern 2 and pattern 5.

Loop2 consists of two patterns: patterns 3 and 4. Let’s see what it tries to find. Loop3 is looking for {any word, with NNP and upperinital or anyword, with NNPS and upperinital} this will match {*Millennium, Sardar, Emirates, Wembley, Ben*} from our example.

Now lets move on to pattern 4, which is repetition of pattern in 3. However notice the presence of “?” at the end meaning optional. Basically, we want to capture the possibility of having two words before the word *“Stadium”*. Using loop 3 and 4 in loop 2 will match further { *Sardar Patel, Ben Hill*} from our examples however *{ Millennium , Emirates, Wembley}* will use ? and escape any matching.

After Loop 2 is run, loop 5 runs, which basically further exploits possibility of another word. Again there is ? to indicate the optional nature of this loop. Hence this time it will match as follows for our running example:

{*Ben Hill Griffin*}, while for others it will be optional.

Hence when the outer loop runs it detects following:
{*Millennium, Sardar Patel, Emirates, Wembley, Ben Hill Griffin}*

Now the rest of the program is straightforward, we are trying to make sure that there is a {Stadium, Circuit, Golf Club} word suffix at the end of this. Pay attention to how the “Golf Club” is implemented, as they are two words unlike other cases.

```
(
        {Token.string =~ "[Ss]tadium"}
        |
        {Token.string =~ "[Cc]ircuit"}
        |
        {Token.string =~ "[Aa]rena"}
        |
        (
        {Token.string =~ "[Gg]olf"}
        {Token.string =~ "[Cc]lub"}
        )

)
```

Refer to this example when you are trying to write a generic grammar which has number of patterns within and you need to make some of the patterns optional.

Exercise 4:

1. Write an example that detects location by using context "at"

For example, at London, UK
at Leeds, Yorkshire
at the Emirates Stadium

For answers see locationcontext2.jape from Example 4 folder.

Example 5. Priority in JAPE rules

For demonstrating the use of priority in JAPE grammar, we will create few competing rules that can identify same entities. For example, we want to identify name of a team based on a few possible contexts. In the first context we use the fact that, in some sports related articles, player names are mentioned closer to name of their teams in the same sentence. For instance, in the following text (Example 5.txt) team name is followed by mention of a player names:

"Manchester United players Park Ji Sung and Jonny Evans are expected to commit their long-term futures in the coming weeks as the English, European and world champions continue to plan for life after Sir Alex Ferguson."

If we have name of the team identified *(Manchester United)* possibly using gazetteer or using POS tags, then we can build a rule surrounding. Following is one way of building such rule:

```
Rule: teamcontext2
Priority:40
(
  (
  {Token.kind==word,Token.category==NNP,Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  )
  |
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
):team
(
  {Token.string=="players"}
  {Player}
  {Token.string=="and" }
  {Player}
)
-->
:team.Team = {rule= "teamcontext-teamcontext2" }
```

JAPE Grammar 7 teamcontext.jape (rule teamcontext2)

Following lines from the rule, establishes the fact that the pattern starts with one or two Noun (NNP) words (i.e. Manchester United, Liverpool)

```
(
  (
  {Token.kind==word,Token.category==NNP,Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  )
  |
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
):team
```

And immediately followed by player names with strings "player" and "and" in the mix.

```
(
  {Token.string=="players"}
  {Player}
  {Token.string=="and"}
  {Player}
)
```

Another way of looking at this problem is to not explicitly tell what these strings could be *("players", "and")* and handle it with just {Token} as done in the following rule:

```
Rule: teamcontext3
Priority:30
(
 (
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
 )
 |
 {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
):team
(
 {Token}
 {Player}
 {Token}
 {Player}
)
-->
:team.Team = {rule= "teamcontext-teamcontext3" }
```

JAPE Grammar 8 teamcontext.jape (rule teamcontext3)

There is one more variety of the context we can use to detect team name here. If we have name of the city identified then we can say that “City + United”, “City + F.C” or “City + FC” will be name of the team. See rule below.

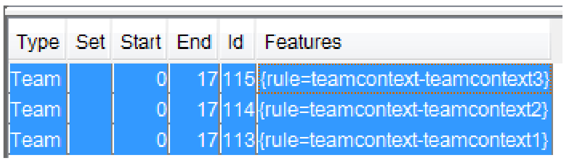
```
Rule: teamcontext1
Priority:50
(

 {City}
      (
       {Token.string=="United" }
       |
       {Token.string=="F.C" }
         |
       {Token.string=="FC" }

      )
):team
-->
:team.Team = {rule= "teamcontext-teamcontext1" }
```

JAPE Grammar 9 teamcontext.jape (rule teamcontext1)

All these competing rules are stored in *teamcontext.jape*. In the *teamcontext.jape* file, by default the control option is set to brill. Let’s look at brill style and what it can do in theory.

The Brill style means that when more than one rule matches the same region of the document, they are all fired. The result of this is that a segment of text could be allocated more than one entity type, and that no priority ordering is necessary. Brill will execute all matching rules starting from a given position and will advance and continue matching from the position in the document where the longest match finishes.

Let's see what Brill style does with our current example. Load the Example 5.txt file from the data store. Run the jape grammar with the transducer in the GATE GUI. You will see that the transducer detects name of the team exactly once through all the three contexts (Figure 3). This is because, all these three rules match the same region of the document, hence transducer fires all three rules. As you might have guessed, we can do without priority ordering here as it had had no effect on the result.



Type	Set	Start	End	Id	Features
Team		0	17	115	{rule=teamcontext-teamcontext3}
Team		0	17	114	{rule=teamcontext-teamcontext2}
Team		0	17	113	{rule=teamcontext-teamcontext1}

Figure 3 Results using Brill style

You can try the same grammar with All by changing the control to *"All"*.

The "all" style is similar to Brill, in that it will also execute all matching rules, but the matching will continue from the next offset to the current one.

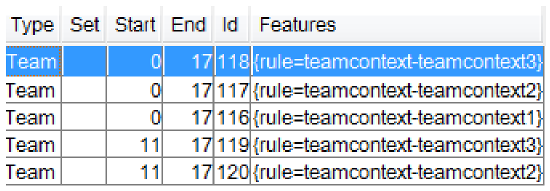
For example, where [] are annotations of type Ann

[aaa[bbb]] [ccc [ddd]]

Then a rule matching {Ann} and creating {Ann-2} for the same spans will generate:

BRILL: [aaabbb] [cccddd]
ALL: [aaa[bbb]] [ccc[ddd]]

To explain this in our example, let's decode following result.



Type	Set	Start	End	Id	Features
Team		0	17	118	{rule=teamcontext-teamcontext3}
Team		0	17	117	{rule=teamcontext-teamcontext2}
Team		0	17	116	{rule=teamcontext-teamcontext1}
Team		11	17	119	{rule=teamcontext-teamcontext3}
Team		11	17	120	{rule=teamcontext-teamcontext2}

Figure 4 Results with All type

The first three results are same as the ones from the *Brill* style (Figure 3). The *All* style also gathers two more annotations (Start=11, End=17, Rule=teamcontext2, String= "United") and (Start=11, End=17, Rule=teamcontext3, String= "United"). Both the rules (teamcontext2 and teamcontext3) have following common pattern that result in those two annotations:

```
(
 (
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
 ) //loop 1
 |
{Token.kind==word, Token.category==NNP, Token.orth==upperInitial} //loop2
):team
```

In the *brill* style,
[Manchester United] matches *loop 1* from the above while [United] for the *loop 2* from the above, hence only selected [Manchester United] the longest match. As Brill will execute all matching rules starting from a given position and will advance and continue matching from the position in the document where the longest match finishes.

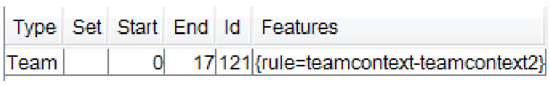
In the *all* style,

[Manchester United] matches *loop 1* from the above while [United] for the *loop 2* from the above, hence first selects [Manchester United] through *loop 1* but goes back to *loop 2* straight after that and also detects [United]. This is because the matching continues from the next offset to the current one.

With the appelt style, only one rule can be fired for the same region of text, according to a set of priority rules. Priority operates in the following way.

1. From all the rules that match a region of the document starting at some point X, the one which matches the longest region is fired.

2. If more than one rule matches the same region, the one with the highest priority is fired
3. If there is more than one rule with the same priority, the one defined earlier in the grammar is fired.



Type	Set	Start	End	Id	Features
Team		0	17	121	{rule=teamcontext-teamcontext2}

Figure 5 Results with Applet option

The longest match is identified here. Infect they are two longest matches that match the same region of the text. Rules *teamcontext2* and *teamcontext3* will select "Manchester United". However, the priority of the rule *teamcontext2* is higher (40) than the *teamcontext3* rule (30) hence only *teamcontext2* gets fired. Try swapping the priority and you will see that *teamcontext3* will get selected this time. Marking both the rules with same priority will fire the rule which is first in the grammar file.

Exercise 3:

1. From the data store, load the *exercise5.txt* file in the GUI. Also load the *playercontext.jape* file from *Example 5* folder. Before you run this Jape grammar, look at the text in *exercise5.txt* and write down what you would expect the result of running the program using *"Brill"*, *"Appelt"* and *"All"*. Run the program and see the results to check against your assumptions and try to understand any differences.
2. The "appelt" control style is the most appropriate for named entity recognition as under "appelt" only one rule can fire for the same pattern. Do you agree?

Example 6. Handling repetitiveness in patterns using Macro

We can use macros with JAPE grammar with the same effect as in the programming languages. In the Example 2 of this tutorial, the reusable pattern `Lookup.majorType == Person` can be converted in a macro. Look for *PersonMacro.jape* file from the folder Example 5 and load it into GUI. Run the jape transducer on Example 6.txt file to inspect the results. You will achieve same results as with the *nestedpattern.jape.*

```
Phase: PersonMacro
Input:  Lookup Token
//note that we are using Lookup and Token both inside our rules.
```

```
Options: control = appelt

Macro:  PERSON
(
      {Lookup.majorType == Person}
)

//trying to detect entities with word "player" mentioned before a
person's name
Rule: playerid
(
 {Token.string == "player"}
)
:temp
(
PERSON
|
 (
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
  {Token.kind==word, Token.category==NNP, Token.orth==upperInitial}
 )
)
:player
-->
 :player.Player= {rule = "playerid"}
```

JAPE Grammar 10 PersonMacro.jape

Example 7. Using negation operator in JAPE

Following shows you how to use a negation operator in JAPE grammar. Let's take one example to demonstrate the requirement of the negation operator in entity extraction. For example, we are looking for titles in the text but particularly not interested in title "*Sir*". The correct rule shall not detect title ("Sir") from the following story text (Example 7.txt) but detect "Mr":

```
Park Ji Sung and Jonny Evans are expected to commit their long-term
futures to Manchester United in the coming weeks as the English,
European and world champions continue to plan for life after Sir Alex
Ferguson.
However Mr Alex Ferguson was unavailable to comment.
```

Rule for doing so is in the folder Example 7, *NegationOperator.jape*.

```
Rule: negationop
(
 {Lookup.majorType == "Title", !Token.string =~ "[Ss]"}
)
:TitleNotStartingWithS
(
{Lookup.majorType == "Person"}
):person
```

```
-->
:TitleNotStartingWithS.Title = {rule= "negationop" },
:person.Person = {rule= "negationop" }
```

JAPE Grammar 11 NegationOperator.jape

The line:

```
{Lookup.majorType == "Title",!Token.string =~ "[Ss]"}
```

Will take care of ignoring the title *"Sir"* and making sure that *person* will be annotated only once as the rule as a whole will be applied.

Example 8. Using JAVA in RHS of JAPE Grammar

The RHS of a JAPE rule can consist of any Java code. This is useful for removing temporary annotations and for percolating and manipulating features from previous annotations identified by the LHS. The example text story *(Example 8.txt)* we are using for this example is:

```
Soccer   - Rooney Gerrard   - File.
Composite  file  picture  of  Liverpool 's  Steven  Gerrard  (left ,
dated   27   September   2006 )   and   Manchester   United 's   Wayne
Rooney   (dated  20  August  2006 ) .  On  the  occasion  of  his
21st  Birthday ,  Tuesday  24  October  2006 ,  Wayne  Rooney  has
hailed  England  team -mate  Steven  Gerrard  as  one  of  the  world
's  best  midfielders  and  wishes  the  Liverpool  star  could  play
at  Manchester  United .
```

We would ideally like to annotate name of the Team with label *"Team"* and also annotate the team name with the property *"teamOfSport"* which is already available through the Lookup.

The JAPE grammar to achieve this is the *usingJAVAinRHS.jape.*

```
Phase:usingJAVAinRHS
Input:  Lookup
Options: control = all

Rule: javainRHS1
(
 {Lookup.majorType == Team}
)
:team
-->
{
gate.AnnotationSet team = (gate.AnnotationSet)bindings.get("team");
gate.Annotation teamAnn = (gate.Annotation)team.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();
features.put("teamOfSport",
teamAnn.getFeatures().get("minorType"));
features.put("rule","javainRHS1");
outputAS.add(team.firstNode(), team.lastNode(), "Team",features);
}
```

JAPE Grammar 12 usingJAVAinRHS.jape

The rule matches a team's name, e.g. *"Manchester United"*, and adds a *teamOfSport* feature depending on the value of the *minorType* from the gazetteer list in which the name was found. We first get the bindings associated with the team label (i.e. the Lookup annotation). We then create a new annotation called *"teamAnn"* which contains this annotation, and create a new *FeatureMap* to enable us to add features. Then we get the *minorType* features (and its value) from the *teamAnn* annotation (in this case, the feature will be *"teamOfSport"* and the value will be *"Football_Club"*), and add this value to a new feature called *"teamOfSport"*. We create another feature *"rule"* with value *"javainRHS1"*. Finally, we add all the features to a new annotation *"Team"* which attaches to the same nodes as the original *"team"* binding.

Note that *inputAS* and *outputAS* represent the input and output annotation set. Normally, these would be the same (by default when using ANNIE, these will be the *"Default"* annotation set) however the user is at liberty to change the input and output annotation sets in the parameters of the JAPE transducer at runtime, it cannot be guaranteed that the input and output annotation sets will be the same, and therefore we must specify the annotation set we are referring to.

Example 9. Using a common file as a holder of application specific JAPE grammar files

So far, we have individual JAPE grammars doing their trick in isolation, however easily we can contemplate real-world scenarios where you want these grammar to work together to achieve a complex task. For achieving this, the list of phases can be specified (in the order in which they are to be run) in a file, conventionally named *main.jape*. When loading the grammar into GATE, it is only necessary to load this main file – the phases will then be loaded automatically. It is, however, possible to omit this main file, and just load the phases individually, but this is much more time-consuming. The grammar phases do not need to be located in the same directory as the main file, but if they are not, the relative path should be specified for each phase.

One of the main reasons for using a sequence of phases is that a pattern can only be used once in each phase, but it can be reused in a later phase. Combined with the fact that priority can only operate within a single grammar, this can be exploited to help deal with ambiguity issues. The solution currently adopted is to write a grammar phase for each annotation type, or for each combination of similar annotation types, and to create temporary annotations. These temporary annotations are accessed by later grammar phases, and can be manipulated as necessary to resolve ambiguity or to merge consecutive annotations. The temporary annotations can either be removed later, or left and simply ignored. Generally, annotations about which we are more certain are created earlier on. Annotations which are more dubious may be created temporarily, and then manipulated by later phases as more information becomes available.

See the difference in the syntax of *main.jape* compared to other jape files that contains single phase.

```
MultiPhase: TestTheGrammars
Phases:
firstname
Fullname
```

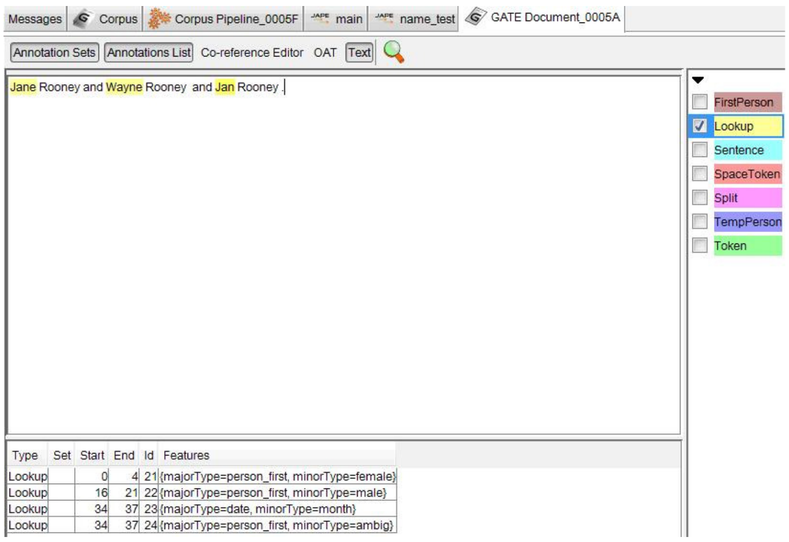
JAPE Grammar 13 main.jape

Example 10. Using JAVA in RHS of JAPE: A complex example

Following is a complex example using Java in RHS. To explain what we are after we will use following text story (Example 10.txt).

"Jane Rooney and Wayne Rooney and Jan Rooney".

The lookup gazetteer annotates the text as following:



Jane Rooney and Wayne Rooney and Jan Rooney .

Type	Set	Start	End	Id	Features
Lookup		0	4	21	{majorType=person_first, minorType=female}
Lookup		16	21	22	{majorType=person_first, minorType=male}
Lookup		34	37	23	{majorType=date, minorType=month}
Lookup		34	37	24	{majorType=person_first, minorType=ambig}

Jane is annotated as *majorType = person_first*, *minorType = female*
Wayne is annotated as *majorType = person_first*, *minorType = male*
Jan is annotated as *majorType = person_first*, *minorType = ambig*

The aim here is to generate full name from this information (Lookup annotation) and at the same time specify gender component of each such person as a property. We are after something like:

Fullname = Jane Rooney , gender = female

The rule will be divided into two phases:

The first phase *(FirstName)* will create an annotation of type *FirstPerson* that basically

> *looks for "Lookup.majorType=person_first" and*
> *creates a gender property by copying minorType values*
> *only if the value is not ambiguous.*

Next phase *(FullName)* will take the *FirstPerson* annotation and will

> Check it the first person is immediately followed by a token which is orthogonally upperInitial if it is then combine them as *Fullname* and create a gender property (by copying from) *FirstPerson* annotation only where it is present

```
Rule: FirstName
// Fred

(
 {Lookup.majorType == person_first}
):person
-->
```

```
{
gate.AnnotationSet                   person                    =
(gate.AnnotationSet)bindings.get("person");
gate.Annotation                  personAnn                     =
(gate.Annotation)person.iterator().next();
gate.FeatureMap features = Factory.newFeatureMap();

//find out if the gender is unambiguous
String gender = (String)personAnn.getFeatures().get("minorType");
boolean ambig = false;
gate.FeatureMap constraints = Factory.newFeatureMap();
constraints.put("majorType", "person_first");
Iterator                       lookupsIter                     =
inputAS.get(personAnn.getStartNode().getOffset()).get("Lookup",
constraints).iterator();
while(!ambig && lookupsIter.hasNext()){
  gate.Annotation anAnnot = (gate.Annotation)lookupsIter.next();
  //we're only interested in annots of the same length

if(anAnnot.getEndNode().getOffset().equals(personAnn.getEndNode().get
Offset())){
    ambig = !gender.equals(anAnnot.getFeatures().get("minorType"));
  }
}
if(!ambig) features.put("gender", gender);

features.put("rule", "FirstName");
annotations.add(person.firstNode(), person.lastNode(), "FirstPerson",
features);
}
```

JAPE Grammar 14 firstname.jape

The example above is more complex, because both the title and the first name (if present) may have a gender feature. There is a possibility of conflict since some first names are ambiguous, or women are given male names (e.g. Charlie). Some titles are also ambiguous, such as “Dr”, in which case they are not marked with a gender feature. We therefore take the gender of the title in preference to the gender of the first name, if it is present. So, on the RHS, we first look for the gender of the title by getting all Title annotations which have a gender feature attached. If a gender feature is present, we add the value of this feature to a new gender feature on the Person annotation we are going to create. If no gender feature is present, we look for the gender of the first name by getting all *FirstPerson* annotations which have a gender feature attached, and adding the value of this feature to a new gender feature on the Person annotation we are going to create. If there is no *FirstPerson* annotation and the title has no gender information, then we simply create the Person annotation with no gender feature.

```
Phase:      Name
Input: FirstPerson Token Lookup
```

```
Options: control = appelt debug = true

////////////////////////////////////////////////////////////
Rule:       Fullname

(
  (
    {FirstPerson}
  )
  (
      {Token.orth == upperInitial}
      |
      {Token.orth == mixedCaps}
  )
)
:person
-->
{
 gate.FeatureMap features = Factory.newFeatureMap();
 gate.AnnotationSet personSet =
(gate.AnnotationSet)bindings.get("person");

HashSet fNames = new HashSet();
fNames.add("gender");

gate.AnnotationSet firstPerson = personSet.get("FirstPerson",fNames);

if(firstPerson != null && firstPerson.size()>0)
{
  gate.Annotation personAnn = (gate.Annotation)
  firstPerson.iterator().next();
  features.put("gender", personAnn.getFeatures().get("gender"));
}
  features.put("kind", "Fullname");
  features.put("rule", "Fullname");
  annotations.add(personSet.firstNode(), personSet.lastNode(),
"Fullname",features);

}
```

JAPE Grammar 15 Fullname.jape

This is how everything makes sense:

For the text:
Jane Rooney and Wayne Rooney and Jan Rooney

Let’s debug the code

In the first scan LHS will pickup

```
(
 (
    {FirstPerson}
  )
  (
      {Token.orth == upperInitial}
      |
      {Token.orth == mixedCaps}
  )
):person
```

Jane	{FirstPerson}
Rooney	{Token.orth == upperInitial}

Now on the right hand side,

```
gate.FeatureMap features = Factory.newFeatureMap();
gate.AnnotationSet personSet =
(gate.AnnotationSet)bindings.get("person");
```

The above two lines will retrieve *personSet* annotations from the LHS side (here it will be Jane Rooney)

```
HashSet fNames = new HashSet();
fNames.add("gender");
gate.AnnotationSet firstPerson = personSet.get("FirstPerson",fNames);
```

Above three lines will find retrieve FirstPerson with gender feature for this iteration. For example it is Jane (gender=female)

Now,

```
if(firstPerson != null && firstPerson.size()>0)
            {
gate.Annotation personAnn = (gate.Annotation)
firstPerson.iterator().next();
features.put("gender", personAnn.getFeatures().get("gender"));
            }
```

First line checks if *firstPerson* is available, if it is then attempts to retrieve gender of the First Person and adds into new feature list. However if there is no gender available, then we still want to create a firstname.

```
features.put("kind", "Fullname");
features.put("rule", "Fullname");
annotations.add(personSet.firstNode(), personSet.lastNode(),
"Fullname",features);
```

The example folder also includes a slightly different version of this program and is for the debugging purpose *(Fullname_Debugmode.jape).*

Example 11. Using Split to control the application of a rule to a single sentence.

We would like to write a grammar rule where the player is identified by its context. Following is an example text (Example 11.txt):

```
Wayne Rooney is a player of Manchester United. He is not of Arsenal.
```

We want to concentrate on the fact that this context is identified by pattern such as "unknown…………of…………..Team", hence concluding that unknown = player, team= Team. However we want to avoid asserting in this particular case that Wayne Rooney's team is Arsenal, as we can overlook this in the grammar and it might pick, Wayne Rooney ….of …Arsenal instead of Wayne Rooney ….of…Manchester United.

Inspect the grammar rule *playercontext.jape* from the example folder. The LHS of the rule is explained here:

```
Rule: playercontext
Priority:50
(//1
 (//2
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
  {Token.kind == punctuation, Token.category == POS}
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
 )
 |
 (//3
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
  {Token.kind == punctuation, Token.subkind == dashpunct}
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
 )
 |
 (//4
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
  {Token.kind == word, Token.category == NNP, Token.orth == upperInitial}
 )
): person1
({Token,!Split})*//6
({Token.string =~ "[Oo]f"})//7
(TEAM):team1//8
-->
```

Loop 1 is the outer loop and will contain the result of pattern matching in the inner loops of 2, 3 and 4. These inner loops establish person names by looking for two consecutive Nouns (with the possibility of having a hyphen - in between them). Hence it will find

```
“Wayne Rooney”
```

Loop 6, will go on till it finds a split or “Of/of” (which is through loop 7).
This will work just fine for removing false positives. We are not going beyond a full stop (hence a statement) for matching the pattern. This is very useful feature to have and will be required to be used in many cases.
Loop 8 will find a team’s name if present immediately after “Of/of”.
RHS of this rule will add “team name” feature with Player.

Example 12. Co referencing

In linguistics, coreference occurs when multiple expressions in a sentence or document have the same referent.

For example, in the sentence "You said you would help me", the two instances of the word you are most likely referring to the same person or group, in which case they are coreferent. Similarly, in "I saw Scott yesterday. He was fishing by the lake," Scott and he are most likely coreferent.

MEANING

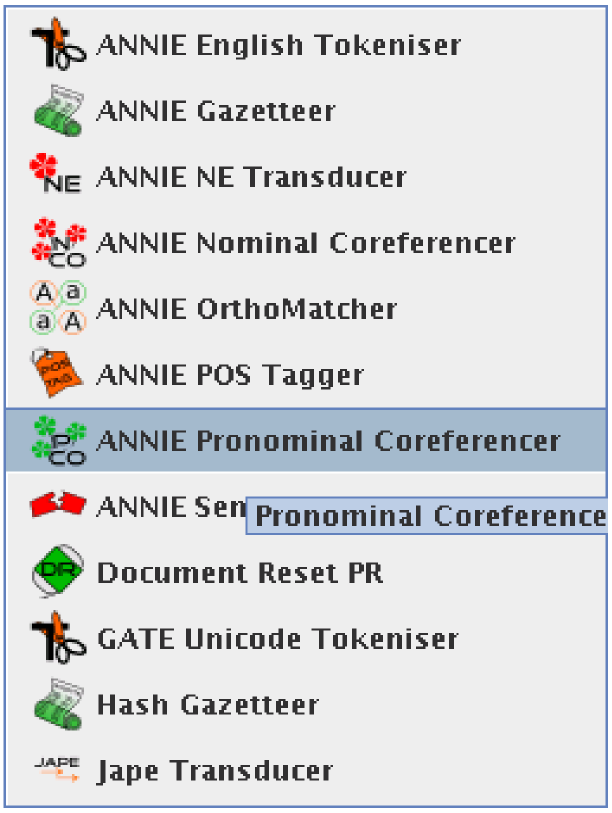
(Source: Glossary of linguistic terms, Book by Eugene E. Loos , Susan Anderson , Dwight H., Day, Jr. , Paul C. Jordan , and J. Douglas Wingate)

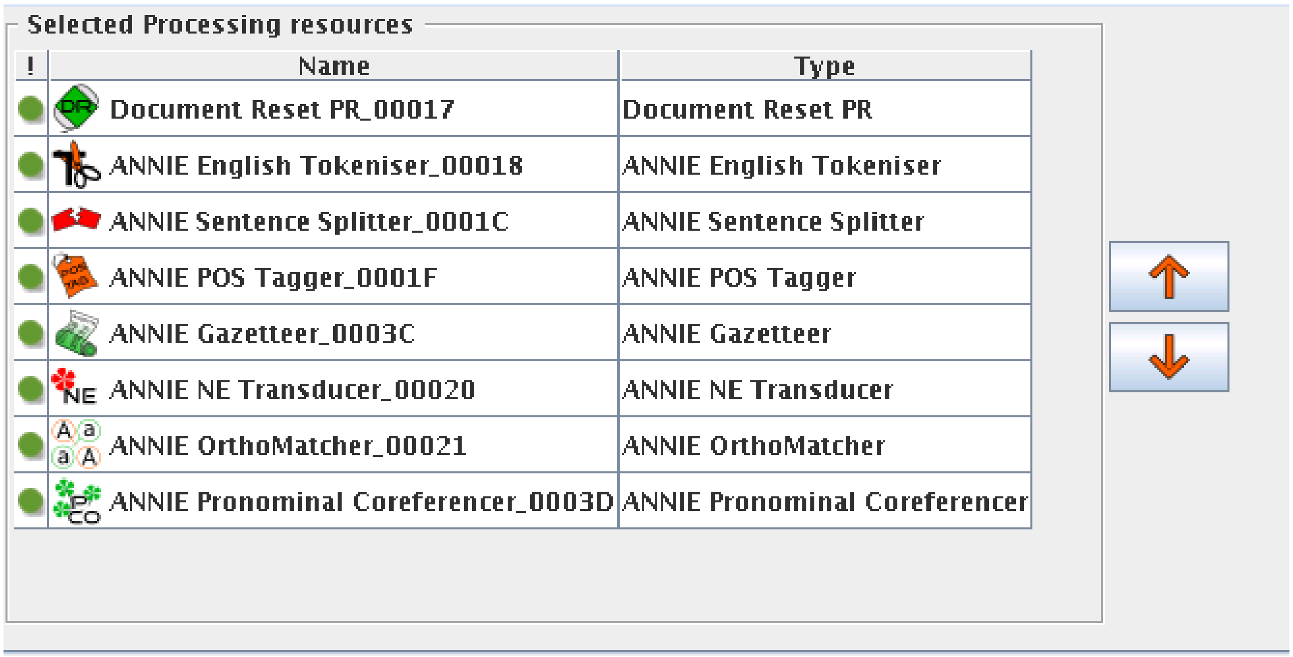
We will use the text (Example 12.txt) .

Wayne Rooney is a player of Manchester United. He is not of Arsenal.

We want to establish that mention of “He” is for “Wayne Rooney” from the first sentence. For this, do the following:

Right click on the Processing Resources Tree and choose Pronominal Coreferencer.



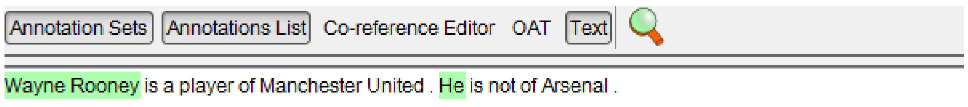

Selected Processing resources

!	Name	Type
●	Document Reset PR_00017	Document Reset PR
●	ANNIE English Tokeniser_00018	ANNIE English Tokeniser
●	ANNIE Sentence Splitter_0001C	ANNIE Sentence Splitter
●	ANNIE POS Tagger_0001F	ANNIE POS Tagger
●	ANNIE Gazetteer_0003C	ANNIE Gazetteer
●	ANNIE NE Transducer_00020	ANNIE NE Transducer
●	ANNIE OrthoMatcher_00021	ANNIE OrthoMatcher
●	ANNIE Pronominal Coreferencer_0003D	ANNIE Pronominal Coreferencer

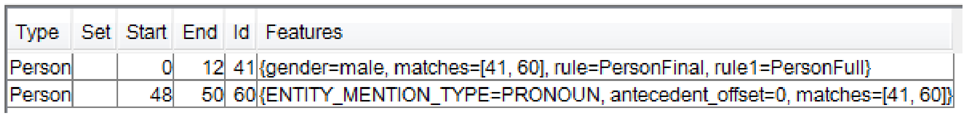
Click Ok

Add it to the end of the pipeline, Ignore the Orthomatcher.

Click Run



Wayne Rooney is a player of Manchester United . He is not of Arsenal .



Type	Set	Start	End	Id	Features
Person		0	12	41	{gender=male, matches=[41, 60], rule=PersonFinal, rule1=PersonFull}
Person		48	50	60	{ENTITY_MENTION_TYPE=PRONOUN, antecedent_offset=0, matches=[41, 60]}

At 48, 50 - *he* has been annotated as a ***Person*** now pointing back to ***Wayne Rooney*** at 0,12

You need to access the annotations of type **Person** and the feature maps associated with it to manipulate and use this information. Following JAPE grammar retrieves this information and passes the features of annotation across referents.

```
Phase: coref
Input:  Lookup Token Split Person
Options: control = brill debug = false

Rule: coref
Priority:50
(
 {Person.ENTITY_MENTION_TYPE == PRONOUN}
):pron
-->
{
gate.AnnotationSet match = (gate.AnnotationSet)bindings.get("pron");
gate.Annotation pronoun = (gate.Annotation)match.iterator().next();

java.lang.Long antOffset =
(java.lang.Long)pronoun.getFeatures().get("antecedent_offset");

gate.AnnotationSet antSet =
(gate.AnnotationSet)annotations.get(antOffset);
gate.Annotation antecedent =
(gate.Annotation)antSet.iterator().next();
int start_offset=0;
int end_offset=0;
String referedEntity = "";
if(antSet != null)
  {
          start_offset =
antecedent.getStartNode().getOffset().intValue();
```

```
            end_offset =
antecedent.getEndNode().getOffset().intValue();
            referedEntity =
doc.getContent().toString().substring(start_offset, end_offset);
  }

String antString = (String)antecedent.getFeatures().toString();
String antType = antecedent.getType();
pronoun.getFeatures().put("refers_to_entity", referedEntity);
pronoun.getFeatures().put("feature of antecedent", antString);
}
```

JAPE Grammar 16 coref.jape

Example 13. Creating Temporary annotations and then deleting at the end when it is no longer useful.

While creating a complex application using GATE, you shall always build on top of temporary annotations which you would like to be deleted at the end of the process.

For example, in the Example 10, we built an annotation *TempPerson (gender)* and then used it to form a full name annotation *FirstName (gender, kind, rule)*. Now, you can write a JAPE grammar which has a RHS allowing you to delete this temporary annotation. This is sort of garbage collection for the system.

```
Phase: Clean
Input: FirstPerson
Options: control = appelt

Rule:CleanTempAnnotations
(
 {FirstPerson}
):temp
-->
{
 gate.AnnotationSet temp = (gate.AnnotationSet)bindings.get("temp");
 annotations.removeAll(temp);
}
```

JAPE Grammar 17 clean.jape

Load the *Example 13.txt* file from the data store and use the *main.jape* to run this grammar file from the Example 13 folder.

Example 14. Creating new entities to use in the JAPE grammar

Let's explain with an example the objective of this exercise. We want to identify a player based on a common natural language pattern such as:

```
"name of the team" "sports_role" "unknown entity"
```

i.e. "England captain *John Terry*" or "Liverpool coach *Raphael Benitez*". The entities to be identified are in italic font. However, rather than writing JAPE rules for each and every sports_role, we can create a gazetteer list and use it for writing JAPE rule.

The gazetteer included with this manual contains such a list, hence the readily annotated text file *Example 14.txt*, contains annotation for sports roles. Look for main.jape in the Example 14 folder and study it to see how the grammar works. This is also an example of how a thread of rules can be sequentially executed to achieve semantically rich entities.

Exercise:

Following text (*Example 14.txt* from the data store) has a pattern "Team1…………..against………Team2" where Team1 and Team2 are part of an Event, playing against each other.

"Liverpool captain Steven Gerrard is confirmed to play against their Real Madrid."

Write a JAPE grammar that has "Team1" and "Team2" available, and it establishes an event relation by making sure that "against" is used to imply an event. For this particular text, the grammar has "Liverpool" and "Real Madrid" as team entities available and the grammar shall establish

Event:

```
{
Team1= Liverpool
Team2 = Real Madrid
}
```

Bibliography

[1] GATE user guide, http://www.gate.ac.uk/sale/tao/split.html
[2] Geoffrey R. Bransford-Koons, Dynamic semantic annotation of California case law, MSc Thesis. http://www-rohan.sdsu.edu/~bransfor/thesis/thesis/index.html, last accessed 10 March, 2009.
[3] Wikepedia
[4] VI editor for windows, http://www.vim.org/download.php
[5] Glossary of linguistic terms, (Book) Eugene E. Loos , Susan Anderson , Dwight H., Day, Jr. , Paul C. Jordan , and J. Douglas Wingate