CS – 00 – 10

JAPE: a Java Annotation Patterns Engine

Hamish Cunningham
Diana Maynard
Valentin Tablan

JAPE: a Java Annotation Patterns Engine
Research memo CS – 00 – 10

JAPE: a Java Annotation Patterns Engine

Hamish Cunningham
Diana Maynard
Valentin Tablan

November 2000
Research memo CS – 00 – 10

Institute for Language, Speech and Hearing (ILASH), and
Department of Computer Science
University of Sheffield, UK

hamish@dcs.shef.ac.uk
http://www.dcs.shef.ac.uk/~hamish

Contents

This specification describes JAPE – a Java Annotation Patterns Engine. JAPE provides finite state transduction over annotations based on regular expressions. JAPE is a version of CPSL – Common Pattern Specification Language[1].

JAPE is available as part of GATE [Cun00, CBTW00, CMB+00, CBPW00].

[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`Doug was a great help to us in implementing JAPE. Thanks Doug!

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. The left-hand-side (LHS) of the rules consist of an annotation pattern that may contain regular expression operators (e.g. `*, ?, +`). The right-hand-side (RHS) consists of annotation manipulation statements. 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.

The first four sections describe the three main modules of JAPE - tokeniser, gazetteer and grammar - and how information is passed between them. Section 5 gives a more formal definition of the JAPE grammar, and some examples of its use. Section 6 describes JAPE's relation to CPSL. The next 3 sections describe the algorithms used, label binding, and the classes used. Section 10 gives an example of the implementation; and finally, section 11 explains the compilation process.

1 Tokeniser

The tokeniser is the first module to be applied. This splits the text into very simple tokens such as numbers, punctuation and words of different types. For example, we might distinguish between words in uppercase and lowercase, and between certain types of punctuation. Although the tokeniser 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.

A rule has a left hand side (LHS) and a right hand side (RHS). The LHS is a regular expression which has to be matched on the input; the RHS describes the annotations to be added to the AnnotationSet. The LHS is separated from the RHS by '>'. The following operators can be used on the LHS:

```
| (or)
* (0 or more occurrences)
? (0 or 1 occurrences)
+ (1 or more occurrences)
```

The RHS uses ';' as a separator, and has the following format:

```
{LHS} > {Annotation type};{attribute1}={value1};...;{attribute
n}={value n}
```

Details about the primitive constructs available are given in the tokeniser file (DefaultTokeniser.Rules).

The following tokeniser rule is for a word beginning with a single capital letter:

```
"UPPERCASE_LETTER" "LOWERCASE_LETTER"* >
  Token;orth=upperInitial;kind=word;
```

It states that the sequence must begin with an uppercase letter, followed by zero or more lowercase letters. This sequence will then be annotated as type “Token”. The attribute “orth” (orthography) has the value “upperInitial”; the attribute “kind” has the value “word”.

2 Gazetteer

The gazetteer lookup module identifies key words in the text which are related to partiualr entity types, by searching a set of pre-stored lists. These gazetteer lists are plain text files, with one entry per line. Each list represents a set of names, such as names of cities, organisations, days of the week, etc.

Below is a small section of the list for units of currency:

```
Ecu
European Currency Units
FFr
Fr
German mark
German marks
New Taiwan dollar
New Taiwan dollars
NT dollar
NT dollars
```

An index file (lists.def) is used to access these lists; for each list, a major type is specified and, optionally, a minor type [2]. In the example below, the first column refers to the list name, the second column to the major type, and the third to the minor type. 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. Grammar rules then specify the types to be identified in particular circumstances.

```
currency_prefix.lst:currency_unit:pre_amount
currency_unit.lst:currency_unit:post_amount
date.lst:date:specific
day.lst:date:day
```

So, for example, if a specific day needs to be identified, the minor type “day” should be specified in the grammar, in order to match only information about specific days; if any kind of date needs to be identified,the major type “date” should be specified, to enable tokens annotated with any information about dates to be identified. More information about this can be found in the section on grammars.

3 Grammar

The JAPE grammar contains rules which act on annotations assigned in earlier phases, in order to produce outputs of annotated entities. The rules are separated into two parts, separated by “–>”. The LHS performs pattern-matching; the RHS describes the annotation to be assigned. On the LHS, the pattern is described in terms of the annotations already assigned by the tokeniser and gazetteer. There are 3 main ways in which the pattern can be specified:

- specify a string of text, e.g. {Token.string == “of”}
- specify the attributes (and values) of a token (or any other annotation), e.g. {Token.kind == number}
- specify an annotation type from the gazetteer, e.g. {Lookup.minorType == month}

[2] it is also possible to include a language in the same way, where lists for different languages are used, though MUSE is only concerned with monolingual recognition

Macros can also be used in the LHS of rules. This means that instead of expressing the information in the rule, it is specified in a macro, which can then be called in the rule. The reason for this is simply to avoid having to repeat the same information in several rules. Macros can themselves be used inside other macros.

The same operators can be used as for the tokeniser rules, i.e.

```
|
*
?
+
```

The pattern description is followed by a label for the annotation. Usually this label would be the same as the attribute value which will be assigned to it, although since the label is only local to the rule, this is for reasons of convenience rather than necessity. It is also possible to have more than one pattern and corresponding label, provided that the pattern is enclosed in a set of round brackets.

The RHS of the rule contains information about the annotation. Information about the annotation is transferred from the LHS of the rule using the label just described, and annotated with the entity type (which follows it). Finally, attributes and their corresponding values are added to the annotation.

In the simple example below, the pattern described will be awarded an annotation of type "Enamex" (because it is an entity name). This annotation will have the attribute "kind", with value "location", and the attribute "rule", with value "GazLocation". (The purpose of the "rule" attribute is simply to ease the process of manual rule validation).

```
Rule: GazLocation
(
{Lookup.majorType == location}
)
:location -->
 :location.Enamex = {kind="location", rule=GazLocation}
```

Grammar rules can essentially be of two types. The first type of rule involves no gazetteer lookup, but can be defined using a small set of possible formats. In general, these are fairly straightforward and offer little potential for ambiguity.

The second type of rules rely more heavily on the gazetteer lists, and cover a much wider range of possibilities. This not only means that many rules may be needed to describe all situations, but that there is a much greater potential for ambiguity. This leads to the necessity for rule ordering and prioritisation, as will be discussed below.

For example, a single rule is sufficient to identify an IP address, because there is only one basic format - a series of numbers, each set connected by a dot. The rule for this is given below[3]:

```
Rule: IPAddress
(
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
 {Token.string == "."}
 {Token.kind == number}
)
:ipAddress -->
 :ipAddress.Address = {kind = "ipAddress"}
```

To identify a date or time, there are many possible variations, and so many rules are needed. For example, the same date information can appear in the following formats (amongst others):

```
 Wed, 10/7/00
 Wed, 10/July/00
 Wed, 10 July, 2000
 Wed 10th of July, 2000
 Wed. July 10th, 2000
 Wed 10 July 2000
```

Different types of date can also be expressed. For example, the following would also be classified as date entities:

```
 the late '80s
 Monday
```

[3]We might be more specific and state the possible lengths of the number, but within the confines of this project we currently have no need to, because there is no ambiguity with anything else

```
St. Andrew's Day
99 BC
mid-November
1980-81
from March to April
```

This also means there is a much greater potential for ambiguity. For example, many of the months of the year can also be girls' Christian names (e.g. May, June). This means that contextual information may be needed to disambiguate them, or we may have to guess which is more likely, based on frequency. For example, while "Friday" could be a person's name (as in "Man Friday"), it is much more likely to be a day of the week.

3.1 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.

For example, the following rule for Time (assuming an appropriate macro for "year") would mean that a year would only be recognised if it occurs preceded by the words "in" or "by":

```
Rule: YearContext1

({Token.string == "in"}|
 {Token.string == "by"}
)
(YEAR)
:date -->
 :date.Timex = {kind = "date", rule = "YearContext1"}
```

Similarly, the following rule (assuming an appropriate macro for "email") would mean that an email address would only be recognised if it occurred inside angled brackets (which would not themselves form part of the entity):

```
Rule: Emailaddress1
({Token.string == ‘‘<’’})
(
 (EMAIL)
)
:email
({Token.string == ‘‘>’’})
-->
 :email.Address= {kind = "email", rule = "Emailaddress1"}
```

3.2 Use of Priority

Each grammar has 2 possible control styles: “Brill” and “Appelt”. This is specified at the beginning of the grammar. 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 proiority ordering is necessary.

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.

An optional priority declaration is associated with each rule, which should be a positive integer. The higher the number, the greater the priority. By default (if the priority declaration is missing) all rules have the priority -1 (i.e. the lowest priority).

For example, the following two rules for location could potentially match the same text.

```
Rule:  Location1
Priority: 25

(
 ({Lookup.majorType == loc_key, Lookup.minorType == pre}
  {SpaceToken})?
 {Lookup.majorType == location}
 ({SpaceToken}
  {Lookup.majorType == loc_key, Lookup.minorType == post})?
)
:locName -->
  :locName.Location = {kind = "location", rule = "Location1"}


Rule: GazLocation
Priority: 20
  (
({Lookup.majorType == location}):location
  )
  -->  :location.Name = {kind = "location", rule=GazLocation}
```

Assume we have the text "China sea", that "China" is defined in the gazetteer as "location", and that sea is defined as a "loc_key" of type "post". In this case, rule Location1 would apply, because it matches a longer region of text starting at the same point ("China sea", as opposed to just "China"). Now assume we just have the text "China". In this case, both rules could be fired, but the priority for Location1 is highest, so it will take precedence. In this case, since both rules produce the same annotation, so it is not so important which rule is fired, but this is not always the case.

One important point of which to be aware is that prioritisation only operates within a single grammar. Although we could make priority global by having all the rules in a single grammar, this is not ideal due to other considerations. Instead, we currently combine all the rules for each entity type in a single grammar. An index file (main.jape) is used to define which grammars should be used, and in which order they should be fired.

4 Walkthrough example

Let us take an example of the whole 3-stage procedure. Suppose we wish to recognise the phrase "800,000 US dollars" as an entity of type "Number", with the feature "money".

First of all, we give an example of a grammar rule (and corresponding macros) for money, which would recognise this type of pattern.

```
Macro: MILLION_BILLION
({Token.string == "m"}|
{Token.string == "million"}|
{Token.string == "b"}|
{Token.string == "billion"}
)

Macro: AMOUNT_NUMBER
({Token.kind == number}
(({Token.string == ","}|
  {Token.string == "."})
{Token.kind == number})*
((SpaceToken.kind == space)?
 (MILLION_BILLION)?)
)

Rule: Money1
// e.g. 30 pounds
  (
      (AMOUNT_NUMBER)
      (SpaceToken.kind == space)?
      ({Lookup.majorType == currency_unit})
  )
 :money  -->
  :money.Number = {kind = "money", rule = "Money1"}
```

4.1 Step 1 - Tokenisation

The tokeniser separates this phrase into the following tokens. In general, a word is comprised of any number of letters of either case, including a hyphen, but nothing else; a number is composed of any sequence of digits; punctuation is recognised individually (each character is a separate token), and any number of consecutive spaces and/or control characters are recognised as a single spacetoken.

```
Token, string = ‘‘800’’, kind = number, length = 3
Token, string = ‘‘,’’, kind = punctuation, length = 1
Token, string = ‘‘000’’, kind = number, length = 3
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘US’’, kind = word, length = 2, orth = allCaps
SpaceToken, string = ‘‘ ’’, kind = space, length = 1
Token, string = ‘‘dollars’’, kind = word, length = 7, orth = lowercase
```

4.2 Step 2 - List Lookup

The gazetteer lists are then searched to find all occurrences of matching words in the text. It finds the following match for the string “US dollars”:

```
Lookup, minorType = post_amount, majorType = currency_unit
```

4.3 Step 3 - Grammar Rules

The grammar rule for money is then invoked. the rule Money1 recognises the entire string “800,000 US dollars” as a Number entity of type Money:

```
Number, kind = money, rule = Money1
```

5 Formal Description of the JAPE Grammar

JAPE is similar to CPSL, with a few exceptions. Figure 1 gives a BNF (Backus-Naur Format) description of the grammar.

An example rule LHS:

```
Rule: KiloAmount
( ({Token.kind == "containsDigitAndComma"}):number
  {Token.string == "kilograms"} ):whole
```

A basic constraint specification appears between curly braces, and gives a conjunction of annotation/attribute/value specifiers which have to match at a particular point in the annotation graph. A complex constraint specification appears within round brackets, and may be bound to a label with the ":" operator; the label then becomes available in the RHS for access to the annotations matched by the complex constraint. Complex constraints can also have Kleene operators (*, +, ?) applied to them. A sequence of constraints represents a sequential conjunction; disjunction is represented by separating constraints with "|".

Converted to the format accepted by the JavaCC LL parser generator, the most significant fragment of the CPSL grammar (as described by Appelt, based on an original specification from a TIPSTER working group chaired by Boyan Onyshkevych) goes like this:

```
constraintGroup -->
    (patternElement)+ ("|" (patternElement)+ )*

patternElement -->
    "{" constraint ("," constraint)* "}"
|   "(" constraintGroup ")" (kleeneOp)? (binding)?
```

Here the first line of `patternElement` is a basic constraint, the second a complex one.

```
MultiPhaseTransducer ::=
   ( <multiphase> <ident> )?
   ( ( SinglePhaseTransducer )+ | ( <phases> ( <ident> )+ ) )
   <EOF>
SinglePhaseTransducer ::=
   <phase> <ident>
   ( <input> ( <ident> )* )?
   ( <option> ( <ident> <assign> <ident> )* )?
   ( ( Rule ) | MacroDef )*
Rule ::=
   <rule> <ident> ( <priority> <integer> )?
   LeftHandSide "-->" RightHandSide
MacroDef ::=
   <macro> <ident> ( PatternElement | Action )
LeftHandSide ::=
   ConstraintGroup
ConstraintGroup ::=
   ( PatternElement )+ ( <bar> ( PatternElement )+ )*
PatternElement ::=
   ( <ident> | BasicPatternElement | ComplexPatternElement )
BasicPatternElement ::=
   ( ( <leftBrace> Constraint ( <comma> Constraint )* <rightBrace> )
     | ( <string> ) )
ComplexPatternElement ::=
   <leftBracket> ConstraintGroup <rightBracket>
   ( <kleeneOp> )?
   ( <colon> ( <ident> | <integer> ) )?
Constraint ::=
   ( <pling> )? <ident> ( <period> <ident> <equals> AttrVal )?
AttrVal ::=
   ( <string> | <ident> | <integer> | <floatingPoint> | <bool> )
RightHandSide ::=
   Action ( <comma> Action )*
Action ::=
   ( NamedJavaBlock | AnonymousJavaBlock |
     AssignmentExpression | <ident> )
NamedJavaBlock ::=
   <colon> <ident> <leftBrace> ConsumeBlock
AnonymousJavaBlock ::=
   <leftBrace> ConsumeBlock
AssignmentExpression ::=
   ( <colon> | <colonplus> ) <ident> <period> <ident>
   <assign>
   <leftBrace> (
     <ident> <assign>
     ( AttrVal | ( <colon> <ident> <period> <ident> <period> <ident> ) )
     ( <comma> )?
   )* <rightBrace>
ConsumeBlock ::=
   Java code
```

Figure 1: BNF of JAPE's grammar

An example of a complete rule:

```
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'.

6 Relation to CPSL

We *differ from the CPSL spec* in various ways:

1. No pre- or post-fix context is allowed on the LHS.
2. No function calls on the LHS.
3. No string shorthand on the LHS.
4. We have two rule application algorithms (one like TextPro, one like Brill/Mitre). See section **??**.
5. Expressions relating to labels unbound on the LHS are not evaluated on the RHS. (In TextPro they evaluate to "false".) See the binding scheme description in section 8.
6. JAPE allows arbitrary Java code on the RHS.
7. JAPE has a different macro syntax, and allows macros for both the RHS and LHS.
8. JAPE grammars are compiled and stored as serialised Java objects.

Apart from this, it is a full implementation of CPSL, and the formal power of the languages is the same (except that a JAPE RHS can delete annotations, which straight CPSL cannot). The rule LHS is a regular language over annotations; the rule RHS can perform arbitrary transformations on annotations, but the RHS is only fired *after* the LHS been evaluated, and the effects of a rule application can only be referenced after the phase in which it occurs, so the recognition power is no more than regular.

7 Algorithms for JAPE Rule Application

JAPE rules are applied in one of two ways:

- in Brill-style, where each rule is applied at every point in the document at which it matches;
- in Appelt-style, where only the longest matching rule is applied at any point where more than one might apply.

In the Appelt case, the rule set for a phase may be considered as a single disjunctive expression (and an efficient implementation would construct a single automaton to recognise the whole rule set). To solve this problem, we need to employ two algorithms:

- one that takes as input a CPSL representation and builds a machine capable of recognizing the situations that match the rules and makes the bindings that occur each time a rule is applied. This machine is a Finite State Machine (FSM), somewhat similar to a lexical analyser (a deterministic finite state automaton).
- another one that uses the FSM built by the above algorithm and traverses the annotation graph in order to find the situations that the FSM can recognise.

7.1 The first algorithm

The first step that needs to be taken in order to create the FSM is to read the CPSL description from the external file(s). This is already done in the old version of Jape.

The second step is to build a nondeterministic FSM from the java objects resulted from the parsing process. This FSM will have one initial state and a set of final states, each of them being associated to one rule (this way we know what RHS we have to execute in case of a match). The nondeterministic FSM will also have empty transitions (arcs labeled with **nil**). In order to build this FSM we will need to implement a version of the algorithm used to convert regular expressions in NFAs.

Finally, this nondeterministic FSM will have to be converted to a deterministic one. The deterministic FSM will have more states (in the worst case s! (where s is the number of states in the nondeterministic one); this case is very improbable) but will be more efficient because it will not have to backtrack.

Let **NFSM** be the nondeterministic FSM and **DFSM** the deterministic one.

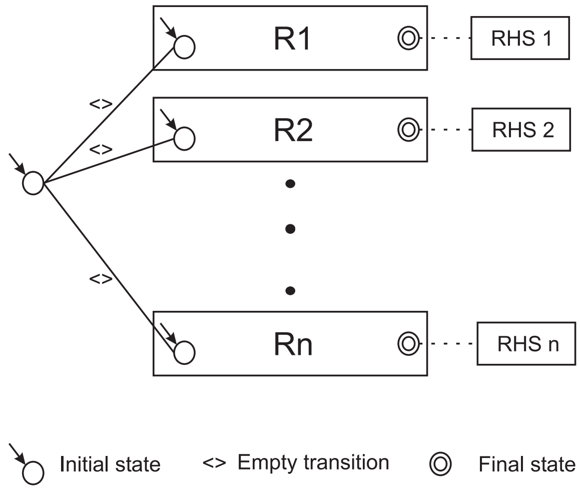

Figure 2: A nondeterministic FSM

The issues that have to be addressed are:

The NFSM will basically be a big OR. This means that it will have an initial state from which empty transitions will lead to the sub-FSMs associated to each rule (see Fig.2). When the NFSM is converted to a DFSM the initial state will be the set containing all the initial states of the FSMs associated to each rule. From that state we will have to compute the possible transitions. For this, the classical algorithm requires us to check for each possible input symbol what is the set of reachable states. The problem is that our input symbols are actually sets of restrictions. This is similar to an automaton that has an infinite set of input symbols (although any given set of rules describes a finite set of constraints). This is not so bad, the real problem is that we have to check if there are transitions that have the same restrictions. I think we can safely consider that there aren't any two transitions with the same set of restrictions. This is safe because if this assumption is wrong, the result will be a state that has two transitions starting from it, transitions that consume the same symbol. This is not a problem because we have to check all outgoing transitions anyway; we will only check the same transition twice.

This leads me to the next issue. Imagine the next part of the transition graph of a FSM:

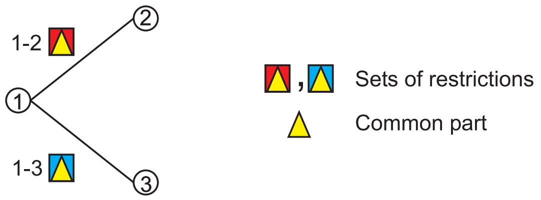

Figure 3: Example of transitions

The restrictions associated to a transition are depicted as graphical figures (the two coloured squares). Now imagine that the two sets of restrictions have a common part (the yellow triangle).

Let us assume that at one moment the current node in the FSM graph (for one of the active FSM instances) is state 1. We get from the annotation graph the set of annotations starting from the associated current node in the annotation graph and try to advance in the FSM transition graph. In order to do this we will have to find a subset of annotations that match the restrictions for moving to state 2 or state 3. In a classical algorithm what we would do is to try to match the annotations against the restrictions "1-2" (this will return a boolean value and a set of bindings) and then we will try the matching against the restrictions "1-3" this means that we will try to match the restrictions in the common part **twice**. Because of the probable structure of the FSM transition graph there will be a lot of transitions starting from the same node which means that may be a lot of conditions checked more than one times.

What can we do to improve this?

We need a way to combine all the restrictions associated to all outgoing arcs of a state (see Fig. 4).

One way to do the (combined) matching is to pre-process the DFSM and to convert all transitions to matchers (as in Fig.4). This could be done using the following algorithm:

- **Input:** A DFSM;
- **Output:** A DFSM with compound restrictions checks.

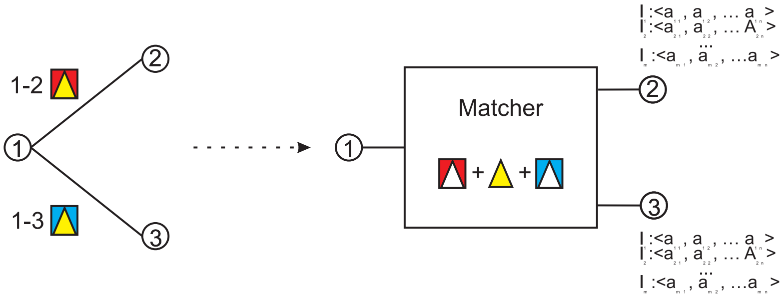

Figure 4: A combined matching process

- for each state s of the DFSM
  1. collect all the restrictions in the labels of the outgoings arcs from s (in the DFSM transition graph)
     Note: these restrictions are either of form "Type == t_1" or of form "Type == t_1 && $Attr_i == Value_i$
  2. Group all these restrictions by type and branch and create compound restrictions of form "[Type == t_1 && $Attr_1 == Value_1$ && $Attr_2 == Value_2$ && ... && $Attr_n == Value_n$]"

     The grouping has to be done with care so it doesn't mix restrictions from different branches, creating unnecessary restrictive queries. These restrictions will be sent to the annotation graph which will do the matching for us. Note that we can only reuse previous queries if the restrictions are identical on two branches. [4]
  3. Create the data structures necessary for linking the bindings to the results of the queries.(see Fig 5)

When this machine will be used for the actual matching the three queries will be run and the results will be stored in sets of annotations (S1..S3 in the picture) and...

- For each pair of annotations from (A_1, A_2) s.t. A_1 in S_1 & A_2 in S_2
  1. a new DFSM instance will be created;

[4]By this we mean restrictions referring to the same type of annotations. If for branches 1-2 and 1-3 the restrictions for the type T_1 are the same, the query for type T_1 will be run only once. Each of the two branches can also have restrictions for other types of annotations.

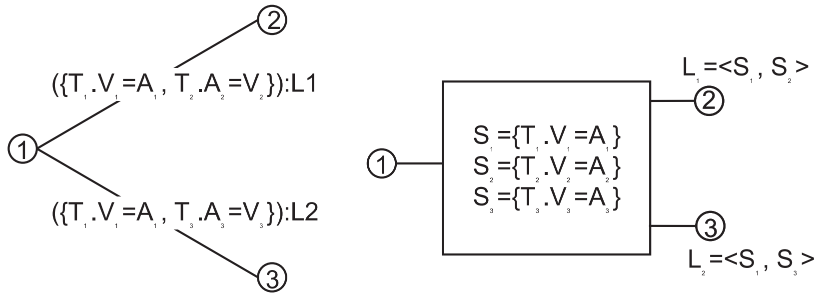

Figure 5: Building a compound matcher

2. this instance will move to state 2;
3. ¡A_1, A_2¿ will be bound to L_1
4. the corresponding node in the annotation graph will become max(A_1 endNode(), A_2.endNode()).

- Similarly, for each pair of annotations from (A_1, A_3) s.t. A_1 in S_1 & A_3 in S_3

1. a new DFSM instance will be created;
2. this instance will move to state 3;
3. ¡A_1, A_3¿ will be bound to L_2
4. the corresponding node in the annotation graph will become max(A_1.endNode(), A_3.endNode()).

While building the compound matcher it is possible to detect queries that depend one from another (e.g. if the expected results of a query are a subset of the results from another query). This kind of situations can be marked so when the queries are actually run some operations can be avoided (e.g. if the less restrictive search returned no results than the more restrictive one can be skipped, or if a search returns an AnnotationSet (an object that can be queried) than the more restrictive query can be.

7.2 Algorithm 2

Consider the following figure:

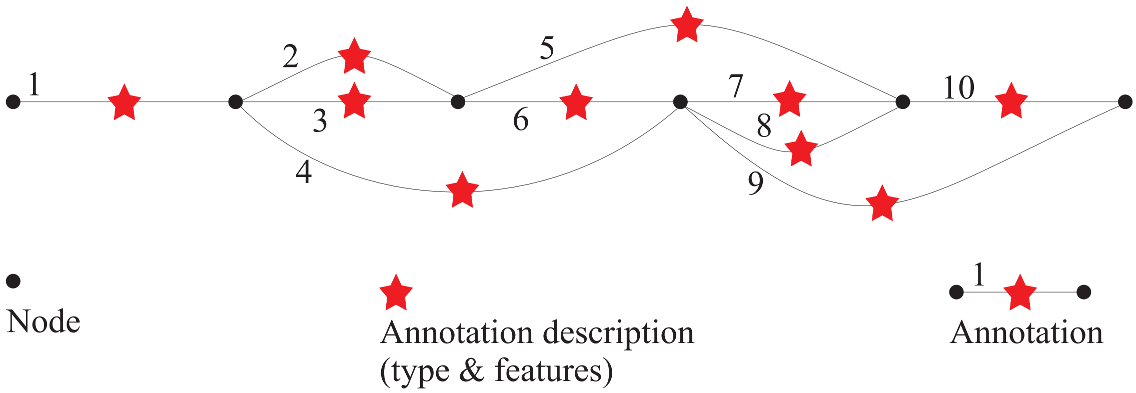

Figure 6: An annotation graph

Basically, the algorithm has to traverse this graph starting from the leftmost node to the rightmost one. Each path found is a sequence of possible matches.

Because more than one annotation (all starting at the same point) can be matched at one step, a path is not viewed as a classical path in a graph, but a sequence of steps, each step being a set of annotations that start in the same node.
e.g. a path in the graph above can be: [1].[2,4].[7,8].[10];
Note that the next step continues from the rightmost node reached by the annotations in the current step.

The matchings are made by a Finite State Machine that resembles an clasical lexical analyser (*aka. scanner*). The main difference to a scanner is that there are no input symbols; the transition from one state to another is based on matching a set of objects (annotations) against a set of restrictions (the constraint group in the LHS of a CPSL rule).

The algorithm can be the following:

1. startNode = the leftmost node
2. create a first instance of the FSM and add it to the list of active instances;
3. for this FSM instance set current node as the leftmost node;
4. while(startNode != last node) do

    1 while (not over) do

1 for each Fi active instance of the FSM do

1 if this instance is in a final state then save a clone of it in the set of accepting FSMs (instances of the FSM that have reached a final state);

2 read all the annotations starting from the current node;

3 select all sets of annotation that can be used to advance one step in the transition graph of the FSM;

4 for each such set create a new instance of the FSM, put it in the active list and make it consume the corresponding set of annotations, making any necessary bindings in the process (this new instance will advance in the annotation graph to the rightmost node that is an end of a matched annotation);

5 discard F_i;

2 end for;

3 if the set of active instances of FSM is empty * then over = true;

end while;

2 if the set of accepting FSMs is not empty

1 from all accepting FSMs select ** the one that matched the longest path;if there are more than one for the same path length select the one with highest priority;

2 execute the action associated to the final state of the selected FSM instance;

3 startNode = selectedFSMInstance.getLastNode.getNextNode();

3 else //the matching failed → start over from the next node // startNode = startNode.getNextNode();

5. end while;

**: the set of active FSM instances can decrease when an active instance cannot continue (there is no set of annotations starting from its current node that can be matched). In this case it will be removed from the set.*

***: if we do Brill style matching we have to process each of the accepting instances.*

The above algorithm is rendered from a semantic point of view. Here are some ideas regarding the actual implementation:

- There is no need to actually create new instances of the FSM. What has to be saved is just the current state of the FSM instance. In this particular case state means:
  - a position in FSM transition graph;
  - a position in the annotation graph (the current node);
  - the set of bindings made during matching annotations from the start node until the current node.
- The most sensitive problem is the method used for *matching*, but this belongs to the other algorithm.

8 Label Binding Scheme

In TextPro a ":" label binds to the last matched annotation in its scope. A "+:" label binds to all the annotations matched in the scope. In JAPE there is no "+:" label (though there is a ":+" – see below), due to the ambiguity with Kleene +. In CPSL a constraint group can be both labelled and have a Kleene operator. How can Kleene + followed by label : be distinguished from label +: ? E.g. given `(....)+:label` are the constraints within the brackets having Kleene + applied to them and being labelled, or is it a +: label?

Appelt's answer is that +: is always a label; to get the other interpretation use `((...)+):`. This may be difficult for rule developers to remember; JAPE disallows the "+:" label, and makes all matched annotations available from every label.

JAPE adds a ":+" label operator, which means that all the spans of any annotations matched are assigned to new annotations created on the RHS relative to that label. (With ordinary ":" labels, only the span of the outermost corners of the annotations matched is used.) (This operator disappears in GATE version 2, with the elimination of multi-span annotations.)

Another problem regards RHS interpretation of unbound labels. If we have something like

```
(
  ( {Word.string == "thing"} ):1
  |
  ( {Word.string == "otherthing"} ):2
)
```

on the LHS, and references to :1 and :2 on the RHS, only one of these will actually be bound to anything when the rule is fired. The expression containing the other should be ignored. In TextPro, an assignment on the RHS that references an unbound label is evaluated to the value "false". In JAPE, RHS expressions involving unbound operators are not evaluated.

9 Classes

The main external interfaces to JAPE are the classes `gate.jape.Batch` and `gate.jape.Compiler`. The CPSL Parser is implemented by `ParseCpsl.jj`, which is input to JavaCC (and JJDoc to produce grammar documentation) and finally Java itself. (There are lots of other classes produced along the way by the compiler-compiler tools:

`ASCII_CharStream.java JJTParseCpslState.java Node.java ParseCpsl.java ParseCpslConstants.java ParseCpslTokenManager.java ParseCpslTreeConstants.java ParseException.java SimpleNode.java TestJape.java Token.java TokenMgrError.java`

These live in the parser subpackage, in the `gate/jape/parser` directory.

Each grammar results in an object of class `Transducer`, which has a set of `Rule`.

Constants are held in the interface `JapeConstants`. The test harness is in `TestJape`.

10 Implementation

10.1 A Walk-Through

The pattern application algorithm (which is either like Doug's, or like Brill's), makes a top-level call to something like

```
boolean matches(int position, Document doc,
                MutableInteger newPosition)
throws PostionOutOfRange
```

which is a method on each `Rule`. This is in turn deferred to the rule's `LeftHandSide`, and thence to the `ConstraintGroup` which each `LeftHandSide` contains. The `ConstraintGroup` iterates over its set of `PatternElementConjunctions`; when one succeeds, the matches call returns true; if none succeed, it returns false. The `Rules` also have

```
void transduce(Document doc) throws LhsNotMatched
```

methods, which may be called after a successful match, and result in the application of the `RightHandSide` of the `Rule` to the document.

`PatternElements` also implement the matches method. Whenever it succeeds, the annotations which were consumed during the match are available from that element, as are a composite span set, and a single span that covers the whole set. In general these will only be accessed via a `bindingName`, which is associated with `ComplexPatternElements`. The `LeftHandSide` maintains a mapping of `bindingNames` to `ComplexPatternElements` (which are accessed by array reference in `Rule RightHandSides`).

Although `PatternElements` give access to an annotation set, these are only built when they are asked for (caching ensures that they are only built once) to avoid storing annotations against every matched element. When asked for, the construction process is an iterative traversal of the elements contained within the element being asked for the annotations. This traversal always bottoms out into `BasicPatternElements`, which are the only ones that need to store annotations all the time.

In a `RightHandSide` application, then, a call to the `LeftHandSide's` binding environment will yield a `ComplexPatternElement` representing the bound object, from which annotations and spans can be retrieved as needed.

10.2 Example RHS code

Let's imagine we are writing an RHS for a rule which binds a set of annotations representing simple numbers to the label `:numbers`. We want to create a new annotation spanning all the ones matched, whose value is an Integer representing the sum of the individual numbers.

The RHS consists of a comma-separated list of blocks, which are either anonymous or labelled. (We also allow the CPSL-style shorthand notation as implemented in TextPro. This is more limiting than code, though, e.g. I don't know how you could do the summing operation below in CPSL.) Anonymous blocks will be evaluated within the same scope, which encloses that of all named blocks, and all blocks are evaluated in order, so declarations can be made in anonymous blocks and then referenced in subsequent blocks. Labelled blocks will only be evaluated when they were bound during LHS matching. The symbol `doc` is always scoped to the Document which the `Transducer` this rule belongs to is processing. For example:

```
// match a sequence of integers, and store their sum
Rule:   NumberSum

( {Token.kind == "otherNum"} )+ :numberList

-->

:numberList{
  // the running total
  int theSum = 0;

  // loop round all the annotations the LHS consumed
  for(int i = 0; i<numberListAnnots.length(); i++) {

    // get the number string for this annot
    String numberString = doc.spanStrings(numberListAnnots.nth(i));

    // parse the number string and add to running total
    try {
      theSum += Integer.parseInt(numberString);
    } catch(NumberFormatException e) {
      // ignore badly-formatted numbers
    }
  } // for each number annot

  doc.addAnnotation(
    "number",
    numberListAnnots.getLeftmostStart(),
    numberListAnnots.getRightmostEnd(),
    "sum",
```

```
    new Integer(theSum)
  );

} // :numberList
```

This stuff then gets converted into code (that is used to form the class we create for RHSs) looking like this:

```
package japeactionclasses;

import gate.*; import java.io.*; import gate.jape.*;
import gate.util.*; import gate.creole.*;

public class Test2NumberSumActionClass
implements java.io.Serializable, RhsAction {

  public void doit(Document doc, LeftHandSide lhs) {

    AnnotationSet numberListAnnots = lhs.getBoundAnnots("numberList");
    if(numberListAnnots.size() != 0) {
      int theSum = 0;

      for(int i = 0; i<numberListAnnots.length(); i++) {
        String numberString = doc.spanStrings(numberListAnnots.nth(i));

        try {
          theSum += Integer.parseInt(numberString);
        } catch(NumberFormatException e) {    }
      }

      doc.addAnnotation(
        "number",
        numberListAnnots.getLeftmostStart(),
        numberListAnnots.getRightmostEnd(),
        "sum",
        new Integer(theSum)
      );

    }
  }
}
```

11 Compilation

JAPE uses a compiler that translates CPSL grammars to Java objects that target the GATE API (and a regular expression library). It uses a compiler-compiler (JavaCC) to construct the parser for CPSL. Because CPSL is a transducer based on a regular language (in effect an FST) it deploys similar techniques to those used in the lexical analysers of parser generators (e.g. lex, flex, JavaCC tokenisation rules).

In other words, the JAPE compiler is a compiler generated with the help of a compiler-compiler which uses back-end code similar to that used in compiler-compilers.

References

[CBPW00] H. Cunningham, K. Bontcheva, W. Peters, and Y. Wilks. Uniform language resource access and distribution in the context of a General Architecture for Text Engineering (GATE). In *Proceedings of the Workshop on Ontologies and Language Resources (OntoLex'2000)*, Sozopol, Bulgaria, September 2000. `http://gate.ac.uk/sale/ontolex/ontolex.ps`.

[CBTW00] H. Cunningham, K. Bontcheva, V. Tablan, and Y. Wilks. Software Infrastructure for Language Resources: a Taxonomy of Previous Work and a Requirements Analysis. In *Proceedings of the 2nd International Conference on Language Resources and Evaluation (LREC-2)*, Athens, 2000. `http://gate.ac.uk/`.

[CMB+00] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, and Y. Wilks. Experience of using GATE for NLP R&D. In *Proceedings of the Workshop on Using Toolsets and Architectures To Build NLP Systems at COLING-2000*, Luxembourg, 2000. `http://gate.ac.uk/`.

[Cun00] H. Cunningham. *Software Architecture for Language Engineering*. PhD thesis, University of Sheffield, 2000. `http://gate.ac.uk/sale/thesis/`.