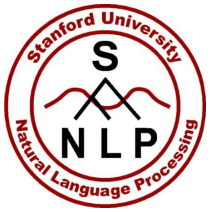
An Introduction to TokensRegex

Angel Xuan Chang

May 30, 2012

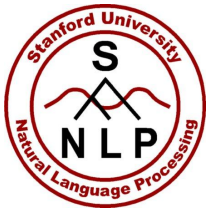


What is TokensRegex?

- A Java utility (in Stanford CoreNLP) for identifying patterns over a list of tokens (i.e. *List<CoreMap>*)
- Very similar to Java regex over Strings except this is over a list of tokens
- Complimentary to Tregex and Semgrex
- Be careful of backslashes
  - Examples assumes that you are embedding the pattern in a Java String, so a digit becomes "\\d" (normally it is just \d, but need to escape \ in Java String)

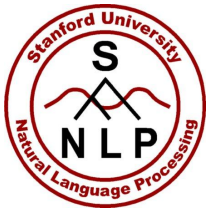
TokensRegex Usage Overview

- TokensRegex usage is like `java.util.regex`
  - Compile pattern
    - `TokenSequencePattern pattern = TokenSequencePattern.compile("/the/ /first/ /day/");`
  - Get matcher
    - `TokenSequenceMatcher matcher = pattern.getMatcher(tokens);`
  - Perform match
    - `matcher.match()`
    - `matcher.find()`
  - Get captured groups
    - `String matched = matcher.group();`
    - `List<CoreLabel> matchedNodes = matcher.groupNodes();`



Syntax – Sequence Regex

- Syntax is also similar to Java regex
- Concatenation: `X Y`
- Or: `X | Y`
- And: `X & Y`
- Quantifiers
  - Greedy: `X+`, `X?`, `X*`, `X{n,m}`, `X{n}`, `X{n,}`
  - Reluctant: `X+?`, `X??`, `X*?`, `X{n,m}?`, `X{n}?`, `X{n,}?`
- Grouping: `(X)`



Syntax – Nodes (Tokens)

- Tokens are specified with attribute key/value pairs indicating how the token attributes should be matched
- Special short hand to match the token text
  - Regular expressions: `/regex/` (use `\/` to escape /)

    To match one or two digits: `/\\d\\d?/`
  - Exact string match: `"text"` (use `\"` to escape ")
    - To match "-": `"-"`
    - If the text only include [A-Za-z0-9_], can leave out the quotes
      - To match **December** exactly: `December`
  - Sequence to match date in December
    - `December /\\d\\d?/ /,/ /\\d\\d\\d\\d/`

Syntax – Token Attributes

- For more complex expressions, we use `[ <attributes> ]` to indicate a token
  - `<attributes> = <basic_attrexpr> | <compound_attrexpr>`
- Basic attribute expression has the form `{ <attr1>; <attr2>…}`
  - Each `<attr>` consist of
    - `<name> <matchfunc> <value>`
  - No duplicate attribute names allowed
  - Standard names for key (see `AnnotationLookup`)
    - word=>CoreAnnotations.TextAnnotation.class
    - tag=>CoreAnnotations.PartOfSpeechTagAnnotation.class
    - lemma=>CoreAnnotations.LemmaAnnotation.class
    - ner=>CoreAnnotations.NamedEntityTagAnnotation.class

Syntax – Token Attributes

- Attribute match functions
  - Pattern Matching: `<name>:/regex/`

    (use `\/` to escape `/`)
    - `[ { word:/\\d\\d/ } ]`
  - String Equality: `<attr>:text` or `<attr>:"text"`

    (use `\"` to escape `"`)
    - `[ { tag:VBD } ]`
    - `[ { word:"-" } ]`
  - Numeric comparison: `<attr> [==|>|<|>=|<=] <value>`
    - `[ { word>100 } ]`
  - Boolean functions: `<attr>::<func>`
    - `EXISTS/NOT_NIL: [ { ner::EXISTS } ]`
    - `NOT_EXISTS/IS_NIL`
    - `IS_NUM` – Can be parsed as a Java number

Syntax – Nodes (Tokens)

- Compound Expressions
  - Compose compound expressions using `!`, `&`, and `|`
  - Use `()` to group expressions
- Negation: `!{X}`
  - `[ !{ tag:/VB.*/ } ]` → any token that is not a verb
- Conjunction: `{X} & {Y}`
  - `[ {word>=1000} & {word <=2000} ]`

  → word is a number between 1000 and 2000
- Disjunction: `{X} | {Y}`
  - `[ {word::IS_NUM} | {tag:CD} ]`

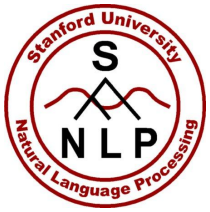
  → word is numeric or is tagged as CD

Syntax – Sequence Regex

- Special Tokens
  - `[]` will match any token
- Putting tokens together into sequences

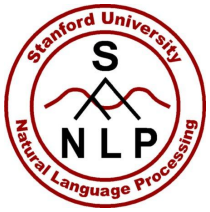
  Match expressions like "from 8:00 to 10:00"
  - `/from/ /\\d\\d?:\\d\\d/ /to/ /\\d\\d?:\\d\\d/`

  Match expressions like "yesterday" or "the day after tomorrow"
  - `(?: [ { tag:DT } ] /day/ /before|after/)? /yesterday|today|tomorrow/`



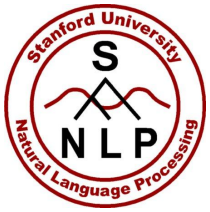
Sequence Regex – Groupings

- Capturing group (default): `(X)`
  - Numbered from left to right as in normal regular expressions
  - Group 0 is the entire matched expression
  - Can be retrieved after a match using
    - `matcher.groupNodes(groupnum)`
- Named group: `(?$name X)`
  - Associate a name to the matched group
    - `matcher.groupNodes(name)`
  - Same name can be used for different parts of an expression (consistency is not enforced). First matched group is returned.
- Non-capturing group: `(?: X)`



Sequence Regex

- Back references
  - Use `\capturegroupid` to match the TEXT of previously matched sequence
- String matching across tokens
  - `(?m){min,max} /pattern/`
  - To match *mid-December* across 1 to 3 tokens:
    - `(?m){1,3) /mid\\s*-\\s*December/`

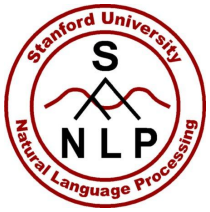


Advanced – Environments

- All patterns are compiled under an environment
- Use environments to
  - Set default options
  - Bind patterns to variables for later expansion
  - Define custom string to attribute key (`Class`) bindings
  - Define custom `Boolean` match functions

Advanced - Environments

- Define an new environment
  - `Env env = TokenSequencePattern.getNewEnv();`
- Set up environment
- Compile a pattern with environment
  - `TokenSequencePattern pattern = TokenSequencePattern.compile(env, …);`



Advanced - Environments

- Setting default options
  - Set default pattern matching behavior
    - To always do case insensitive matching
      - `env.setDefaultStringPatternFlags(Pattern.CASE_INSENSITIVE);`
  - Bind patterns to variables for later expansion
  - Bind pattern for recognizing seasons
    - `env.bind("$SEASON", "/spring|summer|fall|winter/");`
    - `TokenSequencePattern pattern = TokenSequencePattern.compile(env, "$SEASON");`
  - Bound variable can be used as a sequence of nodes or as an attribute value. It cannot be embedded inside the String regex.

Advanced - Environments

- Define custom string to attribute key (`Class`) bindings

```
env.bind("numcomptype",
  CoreAnnotations.NumericCompositeTypeAnn
  otation.class);
```

- Define custom boolean match functions

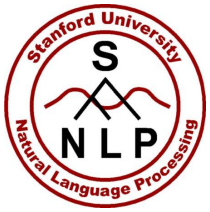
```
env.bind("::FUNC_NAME",
  new NodePattern<T>() {
   boolean match(T in) { … }
  });
```

Priorities and Multiple Patterns

- Can give a pattern priority
  - Priorities are doubles
    - (+ high priority, - low priority, 0 default)
  - `pattern.setPriority(1);`
- List of Patterns to be matched
  - Try the `MultiPatternMatcher` to get a list of non-overlapping matches

```
MultiPatternMatcher<CoreMap> m = new
  MultiPatternMatcher<CoreMap>(patternList);
List<CoreMap> matches =
  m.findNonOverlapping(tokens);
```

  - Overlaps are resolved by pattern priority, match length, pattern order, and offset.



For More Help…

- There is a *JUnitTest* in the TokensRegex package called *TokenSequenceMatcherITest* that has some test patterns
- If you find a bug (i.e. a pattern that should work but doesn't) or need more help, email angelx@cs.stanford.edu

Thanks!