CS 2112 Lab 7: Using Regular Expressions

March 17–19, 2014

Regex Overview

- Regular Expressions, also known as 'regex' or 'regexps' are a common scheme for pattern matching
- regex supports matching individual characters as well as categories and ranges
- A regular expression is represented as a single string and defines a set of matching strings
- Java supports Perl-style regular expressions through `java.util.regex`
- Regex terminology and notation is variable from source to source; almost everything presented here has other names in certain contexts.

Quantifiers

- Quantifiers specify how many of a pattern to match
- `0` matches only the string `0`
- `0*` matches any number of `0`'s, including the empty string
- `0+` matches one or more `0`
- `0?` matches `0` or them empty string
- `0{3,5}` matches `000` or `00000`

Ranges and groups

- Ranges and groups specify a category of characters
- `(1)` is a group and `[1]` is a range.
- `(0|1)` and `[01]` both match `0` or `1`
- `(10)` matches the string `10` but not `1` or `0` alone
- `(ab|cd)` will not match `acbd` but `[abcd]` will
- `[a-z]` matches any lowercase letter
- `[0-9]` matches any digit

Negation

- The ^ character inside a range is the logical negation operator
- `[^0]` matches anything but `0`
- `[^abc]` matches anything but `abc`
- `[^a-z]` matches anything but lowercase letters

Escapes

- regex uses the standard escape sequences like `\n`, `\t`, `\\`
- Characters used in quantifiers and groups must also be escaped
- this includes `\+` `\(` `\.` `\^` among others.
- Interestingly (or annoyingly) `$` is escaped as `$$`

Character Classes

- A character class is a symbol that represents more then one character.
- In most cases the capital letter is the negation of the lowercase
- `\d` = `[0123456789]`, `\D` = `[^0123456789]`
- `\s` matches white space
- `\w` matches a "word", a block of characters surrounded by white space or punctuation.
- `.` matches anything but a newline

Combinations

- Ranges and Quantifiers mix to give useful expressions
- `[a-z]*` matches any number of consecutive lowercase characters
- `[0-9]+` matches all numbers
- `[0-9]3` matches all three digit numbers
- `[A-z]4` matches all four letter words

Chaining

- Multiple combinations start to get at the real power of regex
- `[A-z]1[0-9]1` matches things like `A1`, `B6`, `q0`, etc.
- `[A-Z]1[a-z]* [A-z][a-z]*` matches a properly capitalized first and last name (unless you have a name like `O'Brian` or `McNeil`)
- `[a-z]2,3[0-9]+` matches Cornell net-ids.
- In Java, but not in general, `[ab][cd]` means the union of two ranges, not the intersection.

Java.lang.String

The easiest way to start using regular expressions in Java is through methods provided by the String class. Two examples are `String.split(String)` and `String.replaceAll(String,String)`.

```
1 String TAs = "Reese&Matt&Clara&Ari"; //No offense,Dan
```

```
1 String[] arr = TAs.split("&");
2 for(String s : arr){System.out.println(s);}
```

```
1 System.out.println(TAs.replaceAll("&[^&]+", "&Reese"));
```

Java.util.regex

- More powerful operations are unlocked by the `Java.util.regex` package.
- There are two main classes in this package `Pattern` and `Matcher`
- `Pattern` objects represent regex patterns have a method to return a `Matcher` that allows the pattern to be used.

Java.util.regex.Pattern

- The `Pattern` object has no constructor and instead has a `compile` method that returns a `Pattern` object.
- The Java specific version of regular expressions is documented on the `Pattern` api page, and is well worth reading.
- Note that you must escape your backslashes when coding literals

```
1 Pattern p1 = Pattern.compile("[a-z]{2,3}\\d+");
```

Java.util.regex.Matcher

- `Matcher` does the actual matching work, as the name suggests. Again there is no constructor, but instead a method inside `Pattern` that allows you to get a `Matcher` object set to match on a specific string.
- The principal operations of the `Matcher` are `matches` and `find`. `matches` returns true if the entire string matches the pattern, `find` returns true if any part of the string matches the pattern
- `Matcher` also has methods for operations such as replacement or group capturing.

Replacement example

This example is from the api page:

```
1 Pattern p = Pattern.compile("cat");
2 Matcher m = p.matcher("one cat two cats in the yard");
3 StringBuffer sb = new StringBuffer();
4 while (m.find()) {m.appendReplacement(sb, "dog");}
5 m.appendTail(sb);
6 System.out.println(sb.toString());
```

Capture example

Here is another example this time used to capture a match:

```
1 Pattern p1 = Pattern.compile("([a-z]{2,3}\\d+)@.+");
2 Matcher m = p1.matcher("rpg55@cornell.edu");
3 System.out.println("First group: "+m.group(1));
```

Command line parsing

- Regex can be used to parse command line inputs, capturing can be used to grab the different tags and access them
- Write a calculator using regex that takes commands of the form:
  `num num -f` *or* `num -f num` *or* `-f num num`
  Where `num` represents a positive decimal number (with or without a decimal point) and `-f` is the operation flag, one of `-+ -- -* -/` or `-%`.
- Parse the input and then print the result of the math. Assume no white space pre-parsing.