CS 330 - Fall 2017
Assignment 2, Due: December 10, 2017
Professor: Carlotta Domeniconi

1 POSIX Regular Expression Syntax

Java's POSIX syntax describes regular expressions slightly differently from mathematical tradition (the book follows mathematical tradition).[1] Let's start with the easy stuff:

.	A period matches any character.
*exp******	Matches zero or more *exp*. Greedy: finds long matching substrings. If you are trying to find the longest substring of `abbcyyczz` which matches the regular expression **a.*c** then it will return `abbcyyc` instead of `abbc`.
*exp****?**	Matches zero or more *exp*. Reluctant: finds short matching substrings. If you are trying to find the shortest substring of `abbcyyczz` which matches the regular expression **a.*c** then it will return `abbc`.
*exp***+**	Matches one or more *exp*. Greedy.
*exp***+?**	Matches one or more *exp*. Reluctant.
exp{**5**}	Matches exactly 5 *exp* in a row. Greedy. (obviously you can use numbers other than 5)
exp{**5**}**?**	Matches exactly 5 *exp* in a row. Reluctant.
exp1|*exp2*	Matches either *exp1* or *exp2*.
(?:*exps***)**	Groups the *exps* together into one *exp*. Note that **(***exps***)** also works, but it stores the various *exps* for later use in a special way, wasting memory for our purposes. You probably don't want that. Use **(?:***exps***)**

If you have to pick between reluctant and greedy, often (but not always) you'll want the **reluctant** versions. So let's say we have the string:

```
Four score and seven years ago our fathers brought forth on this
continent a new nation, conceived in liberty, and dedicated to the
proposition that all men are created equal.
```

We're looking for a substring, starting at the beginning, which matches various regular expressions. Here are some regular expressions and the substrings that matched:

[1] egrep differs from both Java and the book as well: likewise the syntaxes used in emacs, vi, Perl, Python, Ruby, etc. You'll have to get used to learning variants for different languages. That's life.

Regex	**Matched...**	
Four score	Four score	
Four.*and	Four score and seven years ago our fathers brought forth on this continent a new nation, conceived in liberty, and	
Four.*?and	Four score and	
.*?s(?:n|v|e)+ ye	Four score and seven ye	*[look at this one carefully]*

Because **(?:n|v|e)** is very inconvenient to write, yet very common, POSIX regular expression syntax has a special way to describe groups of characters where you can pick any of them. We could have instead written **[nve]** which would have simplified the earlier expression to just **.*?s[nve]+ ye** When we write **[nve]** we mean "match *either* the character n *or* the character v *or* the character e". This syntax can only be used for matching individual characters, not strings or more complex expressions.

We can also create ranges. For example, **[a-z]** means "all characters from a to z", and **[a-zA-Z]** means "all characters from a to z and also from A to Z", and **[0-9]** means "all the characters from 0 to 9". Last, and very importantly, we can create negations of characters by putting ^ first. For example, **[^nve]** means "any character *except* for n, v, and e".

How do you write "all the whitespace characters"? Here's one way: **[\t\n\x0B\f\r]** This means "space or tab or linefeed or vertical tab or form-feed or carriage return." Ugh. Since such things are so common, we can instead write: \p{**Space**}

Here are some common shorthand ranges:

\p{**Lower**}	Any lowercase letter, that is, **[a-z]**
\p{**Upper**}	Any uppercase letter, that is, **[A-Z]**
\p{**Alpha**}	Any letter, that is, **[a-zA-Z]**
\p{**Digit**}	Any digit, that is, **[0-9]**
\p{**Alnum**}	Any digit or letter, that is, **[a-zA-Z0-9]**
\p{**Punct**}	Any punctuation
\p{**Blank**}	A space or a tab, that is, **[\t]**
\p{**Space**}	Any whitespace, that is, **[\t\n\x0B\f\r]**

Some more examples. Note that these are all greedy. This is because the * is at the very end, and in that case typically reluctant versions will just match the smallest substring (maybe even empty). You have to play around with this stuff.

Regex	**Matched...**
\p{Alpha}*	Four
F(?:\p{Lower}|\p{Space})*	Four score and seven years ago our fathers brought forth on this continent a new nation
[A-Za-x]*	Four score and seven

There are lots more options, but these will probably suffice for the project. For the full collection of regular expression features in Java, see the documentation for the **java.util.regex.Pattern** object.[2]

2 Using Regular Expressions in Java

Let's say you're trying to match strings of a's and b's. To use the regular expression **a*?b*?a**, you first create a **java.util.regex.Pattern** object. We're going to set it to DOTALL mode, meaning that a period (".") matches even linefeeds:

```
Pattern regex = Pattern.compile("a+?b+?a", Pattern.DOTALL);
```

Now we want to attach it to a string. Let's say the string is **aaaabbaabbbac**. We say:

```
Matcher matcher = regex.matcher("aaaabbaabbbac");
```

You probably want to hold onto the Matcher and reuse it multiple times. Now we can ask interesting questions. For example, does the regular expression match the *entire* string (that is, is the string a member of its language?)

```
matcher.matches();
--> false
```

Okay, how about, is there a substring in the string which matches the regular expression? (This is usually a more interesting question).

```
matcher.find(0);
--> true
```

Now we're cooking! Once we've called `find(0)` we can then ask it where the substring starts, where it ends (that is, the first character after the ending), and what the substring is:

```
matcher.start();
--> 0

matcher.end();
--> 7

matcher.group();
--> "aaaabba"
```

Very interesting! Now, what if we started at position 4 in the string rather than at position 0? (That's the first 'b'). Is there another regular expression which matches?

[2] http://download.oracle.com/javase/6/docs/api/java/util/regex/Pattern.html

```
matcher.find(4);
--> true

matcher.start();
--> 6

matcher.end();
--> 12

matcher.group();
--> "aabbba"
```

Notice that the matcher started at 4, but it didn't begin matching at 4. It wandered up through the string until it found the beginning of a valid regular expression (which started at position 6). So now we have a way to wander through the string looking for substrings, starting at some position, which match the regular expression.

Now let's try this one (note greedy, since the + or * is at the end):

```
Pattern regex = Pattern.compile("[ab]+", Pattern.DOTALL);
Matcher matcher = regex.matcher("aaaabbaabbbac");
matcher.find(0);
--> true

matcher.start();
--> 0

matcher.end();
--> 12

matcher.group();
--> "aaaabbaabbba"
```

How about this one, which should return the first 'a':

```
Pattern regex = Pattern.compile("\p{Alpha}", Pattern.DOTALL);
Matcher matcher = regex.matcher("aaaabbaabbbac");
matcher.find(0);
--> false
```

...Wait, WHAT? The problem is that the backslash character (\) is special in Java strings: it's the escape sequence. To type a backslash you must type *two* backslashes, like this:

```
Pattern regex = Pattern.compile("\\p{Alpha}", Pattern.DOTALL);
Matcher matcher = regex.matcher("aaaabbaabbbac");
```

```
matcher.find(0);
--> true

matcher.start();
--> 0

matcher.end();
--> 1

matcher.group();
--> "a"
```

This can get arbitrarily complicated. Let's say, for example, that you need to match against a string that happens to be two backslashes:

```
\\
```

In a regular expression, since the backslash character is special, you'll need to escape both of them with, you got it, backslashes:

```
\\\\
```

But Java requires that backslashes be escaped in its strings, so to describe these in Java strings, you'll need to escape all four of these! Like this:

```
\\\\\\\\
```

Whew! Some other characters you need to watch out for, escape-wise, when you need to match them:

```
. * + [ ] ( ) | ? ^
```

If you mess up , Java won't issue an error during compile time, but the regular expression pattern system may generate a runtime error informing you of your evil ways.

3 URL Tokens

The project will entail tokenizing (or lexing, or scanning, call it what you will) a URL and breaking it into its constituent parts. URLs can have a fairly complicated syntax[3] but we will be doing just a simple subset. Our URLs will be absolute (non-relative) URLs and will have the following tokens:

[3] Interested? See https://url.spec.whatwg.org/#url-syntax

1. A **protocol**, like `http://` or `file://`
2. An **address**, like `cs.gmu.edu` or `google.com` or `63.88.73.26`. The first (or second) is called a **non-numerical address**, and the third is called a **numerical address**
3. An optional **port**, like `:8080` or `:8443`
4. An optional **file**, like `/~carlotta/` or `/courses/syllabi/index.html` or just `/`
5. An optional **query**, like `?action=edit` or `?promoCode=BC64&utmsource=M1B`
6. An optional **fragment**, like `#url-syntax`

A URL cannot have both a query and a fragment. For fetching, we'll stick with just HTTP. Here are some examples using the above elements:

- `http://cs.gmu.edu:8080/courses/syllabi/inex.html?action=edit`
- `http://cs.gmu.edu/~carlotta#url-syntax`
- `http://cs.gmu.edu/~carlotta`
- `http://cs.gmu.edu/~carlotta/`
- `http://63.8873.26`
- `http://63.8873.26/`
- `http://google.com`
- `http://google.com/nexus?promoCode=BC64&utmsource=M1B`

Let's go over these tokens in detail.

1. Protocols A protocol will start with some number of upper or lower case letters, followed by a colon, then two slashes.

```
http://
file://
ftp://
gopher://
```

2. Numerical Addresses A numerical address is four numbers in sequence separated by periods. The numbers can range from 0 to 255. You are **not required** to restrict them to between 0 and 255 — though it would be really nice if you did — but you *are* required to restrict them to 0 to 999. A number may or may not have leading zeros: thus 01 is perfectly valid, as is 001 or 1.

```
129.174.4.2
129.174.004.002
999.999.2.501
```

(This last version isn't technically proper, but you are permitted to accept it).

3. Non-Numerical Addresses A non-numerical address is one or more strings of characters separated by periods. The valid characters are upper- and lower-case letters, numbers, and hyphens.

```
experts-exchange.com
4chan.org
REDDIT.COM
cs.GMU.edu
swedish.chef.bork.bork.bork.com
```

4. Ports A port is a sequence of one more digits (a positive number) which starts with a colon. By the way, when a port is not specified, it defaults to `:80`

```
:80
:8080
:12345
```

5. Files A file is a sequence of zero or more characters which start with a slash `/`. The characters are not permitted to include pound `#` or question mark `?`. Normally you're not allowed to have whitespace and certain other characters, but we'll allow them.

```
/~carlotta
/
/hello/there/how/are/you/index.html
/holycow/
///////
/@%^&*()
```

6. Queries A query is a sequence of zero or more characters which start with a question mark `?`. Normally you're not allowed to have whitespace and certain other characters, but we'll allow them. Note that even if there is a pound sign `#` inside the query, it doesn't signify the start of a fragment.

```
?
?abcde&de=sadf&asd34
?hello#world
????
```

7. Fragments A fragment is a sequence of zero or more characters which start with a pound sign `#`. Normally you're not allowed to have whitespace and certain other characters, but we'll allow them. Note that even if there is a question mark `?` inside the fragment, it doesn't signify the start of a query.

```
#
#fragment
#whatever?heytheresaquestionmarkhere
####
```

4 The Assignment

You will construct a simple tokenizer class called `URLLexer.java`, which takes an array of regular expression strings (one per token category, in the exact order given above) and a string to tokenize. The class will implement the following methods:

- *The constructor* sets up the tokenizer given the regular expressions.
- `reset(`*string*`)` resets the tokenizer to the beginning of *string*, and sets up any other variables you may need to keep track of, such as current position in the input, the matching index for the token, etc.
- `nextToken()` provides the next token, else null if no more tokens, or when it encounters text in the string which can't be tokenized by any of the regular expressions provided.
- `getMatchingIndex()` returns the index into the array of regular expression strings which matched the token which had recently been returned by `nextToken()`
- `getPosition()` returns current position in the token stream where the next token will be extracted.
- `main(...)` will do the primary code as described later.

You are provided a basic Java file which contains empty versions of these functions, plus some constants. Specifically, you're given each of the tokens, a table of regular expressions (which you must fill out), a table of names for the tokens, and an auxiliary function called `fetch(`*String protocol, String numericalAddress, String nonNumericalAddress, String port, String file, String query, String fragment*`)`, which fetches data from a remote HTTP server given the proper information passed in.

4.1 How Main Should Work

Your `main(...)` function will work as follows. You will repeatedly request a URL by printing "URL: ". Once the user has provided a URL, you will trim it of whitespace, then tokenize it. As you tokenize it you will print out the tokens one by one, including their token types. If you find a duplicate token type, you will FAIL. You will also FAIL if the tokenizer cannot recognize any further tokens but you still have characters left to tokenize. If you manage to finish tokenizing a URL, you will pass the tokens to the `fetch(...)` function provided below. Whenever a failure occurs, you will indicate it, then loop again to request another URL.

4.2 Extra Credit

You get extra credit if your numerical address regular expression properly restricts to a sequence of four numbers, separated by periods, where each number is between 0 and 255 inclusive. Leading zeros are permitted.

4.3 A Few Examples and a Gotcha

The following should all work just fine:

```
http://cs.gmu.edu/
http://cs.gmu.edu
http://google.com/
http://google.com:80/
http://google.com
http://63.88.73.26:80/
http://63.88.73.26
http://63.88.73.26/
http://cs.gmu.edu/faculty
http://cs.gmu.edu/faculty/
http://cs.gmu.edu/faculty/index.html
http://cs.gmu.edu/~carlotta
https://url.spec.whatwg.org/#url-syntax
https://www.google.com/search?q=Regular+Expressions
https://www.google.com/search?q=Regular+Expressions&num=1000
```

```
63.88.73.26http://:80/
```

Hmmm, what's going on with that last one? It works just fine with our tokenizer, and yet it is clearly an invalid URL. Keep in mind that to keep things simple we're just breaking this into tokens according to what they *look like*, not where *they are located relative to one another*. If we wanted to do this correctly, we'd build a much more elaborate finite-state automaton which marched through each section (protocol, address, port, ...) in order and failed when something looked amiss, or something even niftier like a **parser** rather than just a **lexer**. In fact, if you read the URL specification[4] that's exactly what's going on.

SUBMISSION: Submit your code electronically to our TA Prakhar Dogra at pdogra@masonlive.gmu.edu.

This assignment must be performed individually. Group work is NOT allowed. Any deviation from this policy will be considered a violation of the GMU Honor Code

The deadline to submit your code electronically is **MIDNIGHT Sunday, December 10**. Late submissions will be penalized **10 points a day**.

[4] https://url.spec.whatwg.org/#url-syntax