Day 27 – Pattern Lab

Today, we continue building knowledge which will be a foundational portion of the Final Project.

The work you do today builds on the Scanner Lab. You should complete the Scanner Lab before beginning this work, which is a pre-requisite for completing the Final Project.

First, recall Regular Expressions from MA 116 – Discrete Structures. Given an input alphabet, a regular expression will generate strings that are accepted by the language. The three constructs in regular expressions are:

- Concatenation: (ab)
- OR: (a|b)
- Kleene Closure, zero or more repeats: a*

For example, consider an input alphabet of { 0 , 1 }, with a language which is the set of strings of 0's and 1's that ***start*** with 01.
The regular expression would be: 0 1 (0 | 1)*

Next, consider an input alphabet of { 0 , 1 }, with a language which is the set of strings of 0's and 1's that ***end*** with 01.
The regular expression would be: (0 | 1)* 0 1

Finally, for an input alphabet of { 0 , 1 }, with a language that is all set of strings of 0's and 1's that start with 01 ***and*** end with 01.
The regular expression would be: 0 1 (0 | 1)* 0 1

Regular Expressions are frequently used in computer science, especially for searching. We will use them to find email id's and URLs in web-pages. For more on Regular Expressions, https://beginnersbook.com/2014/08/java-regex-tutorial/.

Based on the Java syntax, what would you use for a regular expression to match an email id? For a start, how about one or more "word characters," followed by an @ sign, followed by one or more "word characters," followed by a period, and ending with one or more "word characters."

In Java, there are several predefined character expressions, https://docs.oracle.com/javase/tutorial/essential/regex/pre_char_classes.html .

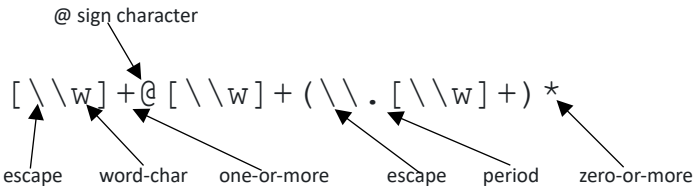
Note the word character definition, `\w`, is `[a-zA-Z_0-9]` . This is translated as lowercase ***a*** thru ***z***, or uppercase ***A*** thru ***Z***, or digits ***0*** thru ***9***.

In Java, since the definition for the word character `\w` starts with a slash, to use it in a program, another `\` is needed in front to indicate that this is a special character.

Since the . is defined as "any character," to use a period, we need `\.` ,and as above, we need to start it with another slash.

Finally, the + operator is defined as "one or more."

Pulling all that together, we get an initial regular expression to match email ids, in Java syntax:



This is a start, but it won't pick up all the email ids.

How would you deploy the above? By declaring and instantiating a `Pattern` Object (pay close attention to the `compile` method)? What class is related to the `Pattern` class? Take note of the Regular Expression constructs.

Read up on Java's `Matcher` class, https://docs.oracle.com/javase/7/docs/api/java/util/regex/Matcher.html. In particular, consider how you would declare and instantiate a `Matcher` Object. Can you do it directly? Do you need to declare and instantiate some type of intermediate Object? What parameter would you use for the constructor?

So, to match, there are several steps. First, define a regular expression `Pattern` object, and `compile` it:

```
Pattern emailPattern = Pattern.compile("your-regular-expression goes here, between the double quotes");
```

Then, this pattern object is used to set up a `Matcher` object:

```
Matcher emailMatch = emailPattern.matcher(token);
```

Finally, the match can be checked with: `urlMatch.find()`

You may copy your working code from the Scanner Lab to a new file, or you can start with the provided file in Moodle, `TDPatternInClass.java`, and modify it as follows:

1. Define a Pattern that matches an email id, the one above is a good start.

2. Update the loop which is reading each token in the URL so that it checks each token for the `emailPattern`, and outputs the result if there is a match.
   You can use the `Matcher` class:
   ```
   Matcher emailMatch = emailPattern.matcher( fill-this-in-with-the-token-you-read-out-of-the-file );
   ```
   and
   ```
   if ( emailMatch.find() )
   {
     String patternFound = emailMatch.group();
   }
   ```

3. Compare the output of your application program to the source code of the URL you submit as input.

   For example, there are 10 emails embedded in the source code for https://jcsites.juniata.edu/faculty/kruse/cs240/PatternLabInput.htm
   And they are:

   1. kruse@juniata.edu
   2. helpe@juniata.edu
   3. LCKruse@yahoo.com
   4. gwk@cfm.brown.edu
   5. kruseb@msn.com
   6. abc@juniata.edu
   7. abc@yahoo.com
   8. Jerry.Kruse@juniata.edu
   9. Jerry-Kruse@juniata.edu
   10. Gerald.Kruse@juniata.edu

4. Improve your regular expression to match more of the embedded email ids. Note that using the above regular expression gives the following output (including labels), so the 8-10 email ids are not parsed correctly:
   ```
   email, number: 1 kruse@juniata.edu
   email, number: 2 helpe@juniata.edu
   email, number: 3 LCKruse@yahoo.com
   email, number: 4 gwk@cfm.brown.edu
   email, number: 5 kruseb@msn.com
   email, number: 6 abc@juniata.edu
   email, number: 7 abc@yahoo.com
   email, number: 8 Kruse@juniata.edu
   email, number: 9 Kruse@juniata.edu
   email, number: 10 Kruse@juniata.edu
   ```

5. Using the above as a template, add functionality to also find URLs embedded in the input page.

   A reasonable first pattern to compile would be:
   `(https?)://[\\w\\.]*([\\w]+)*`

   But note, this won't pick up trailing `/`

   Also, the `?` operator is helpful. `https?` matches all four characters: `http`, and zero (0) or more `s`.