CS 330 Homework
Regexercise

1 Overview

Your responsibility in this homework is to fall in love with regular expressions as a tool for hacking your life. You will do so in the context of writing regexes for a handful of independent pattern-matching and substitution tasks.

Regular expressions are available in many languages. However, in most of these languages, regular expressions are not defined in the syntax of the core language. Rather, they are relegated to a library. Ruby and Perl are notable exceptions, making them a first-class type in the language, complete with their own literal syntax. Many other languages, including Java, overload string literals to compose regular expressions, but since many regex wildcards use backslashes, this overloading forces the developer to do a lot of escaping. We will use Ruby for this assignment.

2 Working with the SpecChecker

The SpecChecker is not meant to be a debugging tool. Make sure you test your code on your own. If the SpecChecker fails on a particular test case, you can continue to test your script directly with a command like the following:

```
./thescript ../specs/grade_regexercise/cases/thecase
```

Do not edit any of the files in the `specs` directory. Your submission will be graded using the unaltered files.

The SpecChecker captures the standard output from your scripts and compares it to the expected output. If you issue debugging print statements to standard output, they will muddy your results. Print to standard error instead:

```
STDERR.puts "First group is #{$1}. Second is #{$2}."
```

3 Requirements

To receive credit for this homework, you must satisfy these requirements:

1. Place all files in directory `<YOUR-REPOSITORY>/regexercise`.
2. Use a Ruby interpreter 2.1 or greater. Run `ruby -v` to check the version. Older versions of Ruby do not support lookaround assertions. I recommend installing a recent version of Ruby with the Ruby Version Manager (RVM) using these steps:

   ```
   gpg --keyserver hkp://keys.gnupg.net --recv-keys 409B6B1796C275462A1703113804BB82D39DC0E3 \
     7D2BAF1CF37B13E2069D6956105BD0E739499BDB
   \curl -sSL https://get.rvm.io | bash -s stable
   rvm install 2.4
   ruby --version
   ```

3. All code must run on a standard Linux machine. If you use another operating system, test your code on a Linux machine before submitting.

4. Since Ruby supports higher-order functions, you should have no need for loops. `Array.each`, `Array.each_with_index`, `Array.map`, `String.scan`, and `gsub` followed by a block should be your primary tools for repetitive processing.

3.1 export

Students in CS 145 inadvertently import non-standard classes, like `com.sun.prism.paint.Color`. These non-standard classes may be installed on the students' machines, but not on the instructor's. Write script `export` to inspect a Java source file and report all non-standard imports. Non-standard here means any package that doesn't start with `java`. Assume that no import statement contains internal comments, but that they may contain arbitrary whitespace in the places where whitespace is allowed. Do not assume that import statements are nicely formatted, one per line.

The path to the Java source file is provided as the only command-line parameter. Print each offending import to standard output on its own line and in the order that they appear in the file. For example, suppose file `Foo.java` contains these lines:

```
package hw1;

import org.twodee.sort.LinearSort;
import java.util.Scanner;
import com.sun.prism.paint.Color;
import android.widget.ListView;
import javax.swing.*;

public class Foo {
  ...
}
```

Running `./export Foo.java` yields the following output:

```
org.twodee.sort.LinearSort
com.sun.prism.paint.Color
android.widget.ListView
```

If a file contains no non-standard imports, exit with status 0. Otherwise, exit with status 1.

3.2 cralf

Close-minded software developers often hardcode platform-dependent linebreaks in their source code. On Windows, these linebreaks are `\r\n`. On Linux and modern macOS, these linebreaks are `\n`. Using platform-dependent linebreaks may cause subtle and annoying issues when someone runs your code on a different operating system.

Write script `cralf` to inspect a source code file and report each occurrence of either of these sequences. The path the source code file is provided as the only command-line parameter. For example, suppose the file `Foo.java` contains this text:

```
public class Foo {
  public static void main(String[] args) {
    System.out.print("foo\n");
    System.out.print("foo\r\n");
    System.out.print("foo\\ab");
  }
}
```

Running `./cralf Foo.java` yields the following output:

```
LF on line 3:     System.out.print("foo\n");
CRLF on line 4:     System.out.print("foo\r\n");
```

If a file contains no platform-dependent linebreaks, exit with status 0. Otherwise, exit with status 1. Assume that the backslashes are not themselves escaped.

3.3 posttruth

Students in CS 145 tend to overuse boolean literals, writing code like `if (isTall == true)` or `else if (isDirectory != false)`. There is never a need to compare to boolean literals[1]. Write a script `posttruth` to inspect a source code file and report each occurrence of a comparison (`==` or `!=`) to a boolean literal. Assume that each such expression appears entirely on one line, and report only the first occurrence on each line. For example, suppose file `Foo.java` contains the following text:

```
class Foo {
  public static void main(String[] args) {
    if (args.length > 0 == true) {
      ...
    }

    if (false == args[0].equals("-n")) {
      ...
    }
  }
}
```

Running `./posttruth Foo.java` yields the following output:

```
Line 3:    if (args.length > 0 == true) {
Line 7:    if (false == args[0].equals("-n")) {
```

If a file contains no comparisons to boolean literals, exit with status 0. Otherwise, exit with status 1.

[1]From Stephen C on StackOverflow: "Would you ask, 'is it true that the Pope is Catholic?' or 'is the Pope Catholic?' ... ?" PHP's and JavaScript's `===` and `!==` operators are an exception to this rule. They ensure that the expression is true and that the operands' types match. So if you want to check for a true boolean, you'd write `if (a === true)`. If you are content with `a` being a non-zero number of a non-null reference, you'd write `if (a)`.

3.4 ymd

Iran, China, Japan, and several other countries have stumbled upon a secret that is plain as day: they format dates in *big endian* order: YYYY/MM/DD. In the United States, we use *middle endian*: MM/DD/YYYY. The advantage of big endian is that one can sort big endian dates chronologically using standard lexicographic (i.e., dictionary order) sorting routines. Sorting middle endian dates requires the date to be parsed and reordered.

Write script `ymd` to read a file whose path is specified as the only command-line parameter and print it to standard output—with all of its middle endian dates switched to big endian. Assume that any sequence of three numbers separated by forward slashes is a middle endian date. Normalize the output date to provide 4 zero-padded digits for the year, 2 zero-padded digits for the month, and the 2 zero-padded digits for the day. For example, suppose file `life.cal` contains the following text:

```
I was born on 6/9/1991. On 10/19/1992 I got my first tattoo.
On 1/1/2001 I married someone I met on lovebefore10.com.
```

Running `./ymd life.cal` yields the following output:

```
I was born on 1991/06/09. On 1992/10/19 I got my first tattoo.
On 2001/01/01 I married someone I met on lovebefore10.com.
```

3.5 footlink

Rule 1503.7 from the fictitious *Decrees of Web Style* states that URLs should never appear in text. Rather, they should be denoted by a numeric footnote reference, and the URL should appear in the corresponding footnote. Write script `footlink` to read a file whose path is specified as the only command-line parameter and print it to standard output—with all of its URLs extracted out to footnotes. For example, suppose file `news` contains the following text:

```
My preferred source for news is http://facebook.com. Before going
out in public, however, I always check http://www.aljazeera.com/
so I don't make a fool of myself. When I need a pick-me-up, I'll
go to https://www.happynews.com.
```

Running `./footlink news` yields the following output:

```
My preferred source for news is [1]. Before going
out in public, however, I always check [2]
so I don't make a fool of myself. When I need a pick-me-up, I'll
go to [3].

[1]: http://facebook.com
[2]: http://www.aljazeera.com/
[3]: https://www.happynews.com
```

For our purposes, a URL is any address that starts with `http://` or `https://` and ends at whitespace or standard English punctuation.

3.6 rationalize

The public policy group Fracternity regularly petitions the government to eliminate numbers with decimal points from public documents, newspapers, and textbooks. They have commissioned you to write plugins for the major browsers to support their mission. Write script `rationalize` to read a file whose path is specified as the only command-line parameter and print it to standard output—with all numbers having digits after the decimal point[2] replaced by a reduced fraction. For example, suppose file `trivia` contains the following text:

```
The desk is 0.5 meters tall.
Abraham Lincon was president for 4.1 years.
I have 5.68 seasons left to watch.
```

Running `./rationalize trivia` yields the following:

```
The desk is 1/2 meters tall.
Abraham Lincon was president for 41/10 years.
I have 142/25 seasons left to watch.
```

For reducing, consider that 5.68 is the same as $5\frac{68}{100}$ or $\frac{568}{100}$. 4 is the largest factor of both the numerator and denominator, so the fraction can be reduced to $\frac{142}{25}$.

3.7 roy

You have a bunch of science-y documents talking about the terahertz frequencies of the visible electromagnetic spectrum. Write script `roy` to annotate all such occurrences with the more familiar color names. Locate all occurrences of the form `NUMBER THz`, where `NUMBER` is in one of the following ranges:

name	*start*	*end*
violet	668	789
blue	606	667
green	526	605
yellow	508	525
orange	484	507
red	400	483

Annotate each occurrence by appending the parenthesized color name. For example, suppose the file `fanbow` contains the following text:

```
My favorite color is 498 THz. Before the accident, Dr. Smith
preferred 551 THz. Now he favors 400 THz.
```

Running `./roy fanbow` yields the following output:

```
My favorite color is 498 THz (orange). Before the accident, Dr. Smith
preferred 551 THz (green). Now he favors 400 THz (red).
```

[2]There's nothing particularly *decimal* about it. *Radix point* is a better name.

3.8 fileach

Some shells, like zshell, provide some excellent file name generation operators. Two in particular are *brace expansion* and *range expansion*. For example, to create files `case_a.txt`, `case_b.txt`, and `case_c.txt`, one can execute `touch case_{a,b,c}.txt`. To edit files `f0` through `f9`, one can execute `vi f<0-9>`. Write script `fileach` to simulate these operators in an environment lacking a good shell.

In your script, provide a function named `expand` that accepts only a string parameter. The parameter is a file name pattern, containing 0 or more brace or range patterns. It first expands all brace patterns from left to right, and then all range patterns from left to right, returning an array of file names. For example, `expand("{a,b,c}<1-2>")` → `["a1", "a2", "b1", "b2", "c1", "c2"]`.

The script itself accepts two command-line parameters: the command to execute on each file (e.g., `touch` or `vi`; `echo` is useful for testing) and the pattern to expand. Use the `system` command and the array returned by `expand` to execute the command on each expanded file.

One may want to reference your `expand` method in another script but not run your main code that lives outside the function. To prevent the main code from being run when `fileach` is imported, structure your `fileach` script like so:

```
def expand
  ...
end

if __FILE__ == $0
  # main code
end
```

`$0` is a builtin variable that is the name of the script that was run by the user. `__FILE__` is a builtin variable that is the name of script currently being interpreted. Only when the two are the same will the main code be run. When `fileach` is imported instead of run directly, `$0` will be the name of the scripting doing the importing.

4 Later Week

To qualify for later-week submission, you must successfully complete the scripts `export`, `cralf`, `posttruth`, and `ymd`.

5 Submission

To submit your work for grading:

1. Run the grading script from your homework directory using `../specs/grade`.
2. Commit and push your work to your repository.
3. Verify that your solution is on Bitbucket by viewing your repository in a web browser.

A passing grading script does not guarantee you credit. Your grade is conditioned on a few things:

- You must meet the requirements described above. The grading script checks some of them, but not all.

- You must successfully submit your code to your repository. Expect to have issues with Git.
- You must not plagiarize. Write your own code. Talk about code with your classmates. Ask questions of your instructor or TA. Do not look at others' code. Do not ask questions specific to your homework anywhere online but Piazza. Your instructor employs a vast repertoire of tools to sniff out academic dishonesty, including: drones, moles, and a piece of software called MOSS that rigorously compares your code to every other submission. You don't want to live in a world serviced by those who squeaked by through questionable means. For your future self, career, and family, do your own work.

The grading script allows you to signal your instructor when requirements are met. You only need to send an email if you qualified for later week submission and are resubmitting after the original deadline.