CS 330: Homework 2

Regexercise: Regular Expressions

1 Overview

Your responsibility in this homework is to fall in love with regular expressions. You will do so in the context of writing a handful of regexes for ten 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 is a notable exception. That we can match patterns without a function call tells us that Ruby's developers think regexes are an important feature of their language.

2 Requirements

In order to complete this homework, please satisfy the requirements below. For each of the required scripts, I parenthetically document how many lines I used in my human-readable, non-cryptic reference implementation. Consider these guidelines, not requirements.

1. Place all files in directory `<YOUR-REPOSITORY>/regexercise`.
2. Use a Ruby interpreter 1.9 or greater. Run `ruby -v` to check the version. If it's not new enough, run `module load all`. (Ruby 1.8 doesn't support look-behind assertions, which make solving some of the problems easier.)
3. All code must run on `dplsubmit`.
4. Only the `calc.rb` script may directly use a loop. `String.scan` and `gsub` should be your primary tools for manipulating the text.
5. Write in `normalize.rb` a script that accepts a path to a text file as its only command-line argument and prints the contents of the file to STDOUT, but with all sequences of 2 or more spaces condensed to a single space. (2 lines)
6. Write in `curlies.rb` a script that accepts a path to a text file as its only command-line argument and prints the contents of the file to STDOUT, but with all curly braces appearing right after their preceding right parenthesis, separated by a single space character. You know what I mean. Some people write code like this:

```
if (isWasteful)
{
  ...
}
```

Replace any form of whitespace characters between `)` and `{` with a single space. (2 lines)

7. Write in `fixlinks.rb` a script that accepts a path to a text file as its only command-line argument and prints the contents of the file to STDOUT, but with all links starting with `www.` and a non-space character, and not yet preceded by `http://`, corrected to include the `http://` prefix. (2 lines)

8. Write in `daynums.rb` a script that accepts a path to a text file as its only command-line argument and prints all the numbers inside the file that could be days in a month—1–31. (If it helps, imagine you are a spy looking for suspicious dates.) Assume that numbers 1–9 may or may not be preceded by a 0. The one or two digits of the numbers must be sandwiched by non-numeric characters. Print each matching number on its own line. (4 lines)

9. Keep your spy hat on. Write in `steg.rb` a script that accepts a path to a text file as its only command-line argument and prints all the first letters of each word to STDOUT in condensed form. For example, if the file contains "Already been chewed," you'll print "Abc". To handle punctuation, remove all non-whitespace, non-letter characters before extracting out the first letters. (2 lines)

10. Write in `serial.rb` a script that accepts a path to a text file as its only command-line argument and prints the contents of the file to STDOUT, but with each occurrence of `#` replaced by its serial number. That is, the first is replaced by 1, the second by 2, and so on. (6 lines)

11. Write in `commas.rb` a script that accepts a number as its only command-line argument and prints the number to STDOUT, but with commas inserted every three digits, starting from the right. For example "1234567" → "1,234,567". (1 line)

12. Write in `attach.rb` a script that accepts a path to a text file as its only command-line argument and returns 1 if the file contains the string "attach" on a line that doesn't start with `>`. Return 0 otherwise. A script's return value is set as the parameter to Ruby's `exit` function. After running the script, `echo $?` to check the returned value. This script could be used in your email client to remind you to attach a file. (6 lines)

13. Write in `calc.rb` a script that evaluates a PEMDAS expression (an expression containing parentheses, exponentiation, multiplication, division, addition, and subtraction) passed as its only command-line argument. It prints the results to STDOUT to six decimal places. For example, "2 ^ (4 - 1.0) + 1" → 9. Parsing such expressions efficiently requires some clever use of data structures. However, a loop, some recursion, and some regexes will work too. Implement this pseudocode, obeying the standard rules of precedence, ignoring whitespace between operators and operands, and using floating point operations:

```
while expr is not yet a plain old number
  if expr contains a parenthesized subexpression
    recursively evaluate the subexpression
    replace the subexpression with its evaluation
  else if the expression contains number ^ number
    replace it with the operation's result
  else if the expression contains number * number or number / number
    replace it with the operation's result
  else if the expression contains number + number or number - number
    replace it with the operation's result
```

You are not allowed to use Ruby's `eval` function in your solution. You may find the builtin variables `$``, `$&`, and `$'` helpful.

To make the recursion possible, your code will need to have a name by which it can be invoked, i.e., you need to make a function. Functions in Ruby have the form:

```
def myFunc(param1, param2)
  statement1
  statement2
  return value
end
```

(22 lines)

14. Write in `markup.rb` a script that accepts a path to a text file as its only command-line argument and prints the contents of the file to STDOUT, but with the text marked up with some HTML according to these rules:

    (a) Each line beginning and ending with one asterisk gets surrounded with `<h1>` and `</h1>`, instead of the asterisks and surrounding whitespace.

    (b) Each line beginning and ending with two asterisks gets surrounded with `<h2>` and `</h2>`, instead of the asterisks and surrounding whitespace.

    (c) Each line beginning with `-` gets surrounded with `<li>` and `</li>`, with the hyphen and leading whitespace removed.

    (d) Each run of `<li>` lines, with each line separated from the next by a single newline, gets surrounded by `<ul>\n` and `\n</ul>`.

    (6 lines)

3 Submission

This homework is graded by hand and with help from the grading script, which can be run from your homework directory with `../specs/grade`. Your assignment is expected to fully meet the requirements above, those checked by the grading script, and the following:

1. Variable names should be meaningful and accurate.
2. Non-obvious parts of your code should be commented.
3. Code should be cleanly formatted and indented.

Your work will also be inspected for plagiarism. Please do your own work. 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. (If you find violators of this rule, please let me know.) Write your own code.

The grading script sends your instructor an email when it successfully completes. All that remains is for you to push your code to Bitbucket.