CSC 372, Spring 2015
Assignment 5
Due: Thursday, March 26 at 23:59:59

Use Ruby 1.9.3!

Use Ruby 1.9.3 for all Ruby assignments. Use "`rvm 1.9`" each time you login on lectura to set things for 1.9.3. You might get a blat about "`Warning! PATH is not properly set up...`" but if `ruby --version` shows 1.9.3, things should be fine. Here's what I see:

```
$ rvm 1.9
Warning! PATH is not properly set up
...lots more...
$ ruby --version
ruby 1.9.3p484 (2013-11-22 revision 43786) [x86_64-linux]
```

Use the tester!

Don't just "eyeball" your output—use the tester! I'll be delighted to help you with using the tester and understanding its output. However, I won't have any sympathy for those who fail tests during grading simply because they didn't want to bother with the tester

Make symbolic links for `a5` and `t` in your assignment 5 directory on lectura, for easy access to the tester and the data files. (And, the tester assumes the `a5` symlink is present.)

Before you make that a5 symlink, let's do one more handy thing, assuming your 372 directory is `~/372`:

```
$ cd ~/372
$ ln -s /cs/www/classes/cs372/spring15 www
```

With that `www` symlink in place you no longer need to pound out `/cs/.../spring15` to make those `aX` links. Just reference `372/www` instead:

```
$ cd a5
$ ln -s ../www/a5 a5
$ ln -s a5/tester t
```

Historical note: Symbolic links were added to 4.2BSD UNIX by Kirk McKusick in the early 1980s. At a Boston USENIX conference he said that "`namei()`" was his "favorite routine". It did the complex job of traversing paths and following symbolic links to eventually reach an "i-node". To provide the full generality that symbolic links have, `namei()` was pretty hard to get right.

Output from command-line examples is followed by a blank line

Just as in the assignment 3 write-up, output from command-line examples in this write-up is followed by a blank line.

Problem 1. (4 points) `nzip.rb`

Write a Ruby iterator `nzip(a1, a2, ..., aN)` that yields a series of arrays. The first array is `[a1[0], a2[0],...aN[0]]`, the second array is `[a1[1], a2[1],...aN[1]]`, etc. `nzip` produces values as long as all arguments have an element at a given position—the shortest array

determines how many results `nzip` will yield. `nzip` returns the number of arrays that it produced. If called with no arguments, `nzip` returns `nil`.

There's a minor restriction on `nzip`: You can't use `Array#zip`!

```
>> a1 = [1,2,3,4]

>> a2 = %w{one two three four five six}

>> a3 = [10] * 10

>> nzip(a1,a2) { |x| printf("result: %s\n", x.inspect) }
result: [1, "one"]
result: [2, "two"]
result: [3, "three"]
result: [4, "four"]
=> 4

>> nzip(a1,a2,a3) {|x| printf("%s, size = %d\n", x.inspect,
x.size)}
[1, "one", 10], size = 3
[2, "two", 10], size = 3
[3, "three", 10], size = 3
[4, "four", 10], size = 3
=> 4

>> nzip([String]) { |x| printf("x = #{x.inspect}\n") }
x = [String]
=> 1

>> nzip([ ], a2) { |x| puts "oops!" }
=> 0
```

Remember that you can't use `Array#zip`!

Note that `nzip` is not a whole program. It is a freestanding method. You can test it with `irb`, using `-r` to load it:

```
$ irb -r ./nzip.rb
>> nzip(("a".."z").to_a, (1..5).to_a) {|x| p x }
["a", 1]
["b", 2]
["c", 3]
["d", 4]
["e", 5]
=> 5
```

Here's maybe a better way to approach the edit-run cycle with `nzip` and the two iterators that follow, `vrepl` and `mirror`:

Add the following line to your `~/.irbrc`:

```
$nz="nzip.rb"
```

After restarting `irb` you can do this:

```
$ irb
>> load $nz
=> true
```

That works because load is a Ruby function. That contrasts with `:load` in `ghci`, which is an operation provided by the REPL, not the Haskell language.

A further step, for those who don't like to watch themselves type, might be an `lnz` function in your `.irbrc`:

```
def lnz; load $nz; end
```

An intermediate step is to put "`alias l load`" in your `.irbrc` and then do this:

```
$ irb
>> l $nz
=> true
```

And/or, you might test with a simple program:

```
$ cat nziptest.rb
load "nzip.rb"
nzip([1,2]) { puts "x" }

$ ruby nziptest.rb
x
x
```

Problem 2. (4 points) `vrepl.rb`

Write a Ruby iterator `vrepl(a)` that produces an array consisting of varying numbers of repetitions of values in `a`. The number of repetitions for each element is determined by the result of the block when the iterator yields that element.

```
>> vrepl(%w{a b c}) { 2 }
=> ["a", "a", "b", "b", "c", "c"]

>> vrepl(%w{a b c}) { 0 }
=> []

>> vrepl((1..10).to_a) { |x| x % 2 == 0 ? 1 : 0 }
=> [2, 4, 6, 8, 10]

>> i = 0
=> 0

>> vrepl([7, [1], "4"]) { i += 1 }
=> [7, [1], [1], "4", "4", "4"]
```

If the block produces a negative value, zero repetitions are produced:

```
>> vrepl([7, 1, 4]) { -10 }
=> []
```

Like `nzip`, `vrepl` is not a program. It is a freestanding method.

Problem 3. (4 points) `mirror.rb`

Write a Ruby iterator `mirror(x)` that yields a "mirrored" sequence of values based on the values that `x.each` yields. Unlike `nzip` and `vrepl`, which operate on arrays, all that `mirror` requires is that `x` responds to the method `each`. (Duck typing!) The value returned by `mirror` is always `nil`.

```
>> mirror(1..3) { |v| puts v }
1
2
3
2
1
=> nil

>> mirror([1, "two", {a: "b"}, 3.0]) { |v| p v }
1
"two"
{:a=>"b"}
3.0
{:a=>"b"}
"two"
1
=> nil

>> mirror({:a=>1, :b=>2, :c=>3}) {|x| p x}
[:a, 1]
[:b, 2]
[:c, 3]
[:b, 2]
[:a, 1]
=> nil

>> mirror([]) { |v| puts v }
=> nil
```

Like `nzip` and `vrepl`, `mirror` is a freestanding method.

Problem 4. (12 points) `calc.rb`

Write in Ruby a simple four-function line-oriented calculator that evaluates expressions composed of integer literals and variables, providing addition, subtraction, multiplication, and division. All operators have equal precedence. Evaluation is done in a strict left to right order. Control-D exits the program. Here are examples of expressions involving integer literals:

```
$ ruby calc.rb
? 3+4
7
? 3*4+5
17
? 3+4*5          Note that the addition is done first because it is the leftmost operator.
35
? 1/2*3+4
4
```

```
? 5/3
1
? 143243243243242323*342343443234324
49038385111943393068867603094652
? ^D
$
```

Variables are created with assignments. Variables begin with a letter and are followed by zero or more letters or digits. Variables have a default value of zero. The result of an assignment is the value assigned.

```
$ ruby calc.rb
? x=7
7
? yval=10
10
? z
0
? x=x+yval+z
17
? yval=x+yval
27
? yval
27
? big=1111111111111111*1111111111111111
1234567901234567876543209876543 21
? big=big/big
1
```

Assignments only appear as the first operation on a line and consist of a variable followed by an equals sign followed by an expression. You won't see something like `x=y=3` or `x+y=0`.

Note that while the arithmetic operators are done in strict left-to-right order, the assignment, if any, is done last.

Input lines consist solely of letters, digits, and the five symbols `+*-/=`. Assume all expressions are well formed. You won't see something like `x==3` or `+10/5+`. If a string starts with a letter, it is a variable; you won't see something like `15x`. There is no negation; you won't see something like `x=-10`. Division by zero is not supported. There will be no empty lines in the input.

*Implementation note*s

Use `String#scan` with a *regular expression* to break up input lines. Here's an example of `scan`:

```
>> "x2=3*val+40-500".scan(/\w+|\W+/)
=> ["x2", "=", "3", "*", "val", "+", "40", "-", "500"]
```

As of press time we're just starting into regular expressions. You can simply use the argument to `scan` that's shown above. Think of it as a black box for now.

Using `to_i` to look for integers like you did on `xfield` won't be good enough for `calc` because an expression might include a 0. You can use `"0" <= c && c <= "9"` to see if `c` is digit, or look ahead a little in the material on regular expressions.

`Kernel#eval` might look like a quick shortcut to a solution for this problem but using it can lead to

some headaches. My recommendation is that you avoid `eval` for this problem. However, I do recommend that you use `Object#send`! It works like this:

```
>> 4.send("+", 3)
=> 7

>> 10.send("-", 4)
=> 6
```

Problem 5. (25 points) `switched.rb`

The U.S. Social Security Administration makes available yearly counts of first names on birth certificates back to 1885. Over time, some names change from predominantly male to predominantly female or vice-versa. For this problem you are to create a Ruby program `switched.rb` to look for names that change from predominantly male to predominantly female in given spans of years.

`switched.rb` takes two command-line arguments: a starting year and an ending year. Here's a run:

```
% ruby switched.rb 1951 1958
             1951   1952   1953   1954   1955   1956   1957   1958
Dana         1.19   1.20   1.26   1.29   1.00   0.79   0.67   0.64
Jackie       1.40   1.29   1.14   1.13   1.11   0.94   0.72   0.57
Kelly        4.23   2.74   3.73   2.10   2.32   1.77   0.98   0.51
Kim          2.58   1.82   1.47   1.08   0.61   0.30   0.17   0.12
Rene         1.43   1.32   1.15   1.24   1.13   0.88   0.87   0.89
Stacy        1.06   0.81   0.62   0.47   0.44   0.36   0.29   0.21
Tracy        1.51   1.14   1.02   0.73   0.56   0.55   0.59   0.59
```

First, note that all numbers in the leftmost column are greater than zero and all numbers in the rightmost column are less than zero.

The 1.19 for Dana in 1951 indicates that in 1951 there were 1.19 times as many male babies named Dana as there were female babies named Dana. We can see that in `a5/yob/1951.txt`, which has the 1951 data:

```
$ grep Dana, a5/yob/1951.txt
Dana,F,1076
Dana,M,1277
```

The format of the `a5/yob/`*`YEAR`*`.txt` dat files is simple: each line has the name, sex, and the associated count, separated by commas.

Note that the argument to `grep`, "`Dana,`" has a trailing comma so that "Danae" doesn't turn up, too.

By 1958 things had changed—there were only .64 males named Dana for every female named Dana:

```
$ grep Dana, a5/yob/1958.txt
Dana,F,2388
Dana,M,1531
```

`switched.rb` reads the `a5/yob/`*`YEAR`*`.txt` files for all the years in the range specified by the command line arguments and looks for names for which the male/female ratio is > 1.0 in the first year and < 1.0 in the last year. For all the names it finds, it prints the male/female ratio for all the years from the

first year through the last year. Names are printed in alphabetical order.

As a specific example, Dana is included in the list for 1951 through 1958 because males/females in 1951 was 1.19 (> 1.0) and males/females in 1958 was 0.64 (<1.0). The ratios for the middle years are not examined to decide whether to include a name; they are shown only to provide a more complete picture of the data between the endpoints.

Note that there's a big shift for Kim from 1954 through 1957. I wonder if that because the actress Kim Novak had a breakout role in 1955's *Picnic*.

If no names meet the criteria, `switched` prints "`no names found`" and exits by calling `exit`.

```
$ ruby switched.rb 2011 2012
no names found
```

IMPORTANT: To eliminate the less significant results, a name is included only if both the male and female counts in both the first and last year is greater than or equal to 100. By that criteria the name Lavern is not included for 1949-1951, and no other names turn up, either:

```
% ruby switched.rb 1949 1951
no names found
```

Here's the underlying data:

```
$ grep Lavern, a5/yob/yob1949.txt
Lavern,F,93
Lavern,M,121

$ grep Lavern, a5/yob/yob1951.txt
Lavern,F,95
Lavern,M,86
```

There was a M/F shift from 121/93 in 1949 to 86/95 in 1951 but because not all four of those counts are >= 100, Lavern is not included. There's no `grep` shown above for 1950 because that data is inconsequential: inclusion is determined solely based on counts for the first and last year.

It's interesting to combine `switched` with a `bash` `for`-loop that runs the program with a gradually shifting range. Here's the command, in case you'd like to try it:

```
for i in $(seq 1940 2002); do ruby switched.rb $i $(($i+9)); echo ===; done
```

Two obvious extensions to `switched` would be command-line options to adjust the 100-baby minimum and to look for female to male flips for a name. You might find those interesting to implement and experiment with, but neither are required.

`switched.rb` does no error handling whatsoever. Behavior is only defined in the case of being given two command line arguments in the range of 1885 to 2012, and the first must be less than the second.

Implementation notes for `switched`

Use `File.open` to produce a `File` object whose `gets` method can be used to read lines. Example:

```
$ cat fileio.rb
```

```
year = ARGV[0]

f = File.open("a5/yob/#{year}.txt")

count = 0
while line = f.gets
    count += 1
end

f.close

puts "read #{count} lines"

$ ruby fileio.rb 2001
read 30258 lines
```

Alternatively, you could use `f.readlines()` to produce an array of all the lines in the file with a single call or `f.each { ... }` to process each line with the associated block.

The M/F ratios are formatted using a `%7.2f` format with `printf`, demonstrated on the command line with `ruby -e:`

```
$ ruby -e 'printf("%7.2f\n", 1277.0/1076.0)'
  1.19
```

Names are left-justified in a 10-wide field using a `printf` format of `%-10s`.

You may have questions about the data files. Before asking me or posting on Piazza, take a look at the data files and see if you can answer the question yourself. The files are in `a5/yob`. Those same files will be used for testing when grading.

You can download http://www.cs.arizona.edu/classes/cs372/spring15/a5/yob.zip for testing on your own machine. In the same directory as your `switched.rb`, make a directory named `a5` and then unzip `yob.zip` in that directory to produce a structure compatible with the `File.open` above.

I intend this problem to be an exercise in using the `Hash` class. I encourage you to devise a data structure yourself but in case you run into trouble, here are a few thoughts on my approach to the problem: http://www.cs.arizona.edu/classes/cs372/spring15/a5/switched-hint.html

Problem 6. (2 points) `twoways.txt`

Ruby has two entirely different ways to specify blocks. One form encloses the expressions of the block in curly braces. The other form uses `do...end`. For this problem I'd like you to address two questions:

(1) Why do you think that Ruby has both forms?
(2) Present an argument that is either in favor of keeping both forms or removing one from the language. If you're in favor of removing one, be sure to identify the one you'd like to get rid of.

No Googling on this one, please. Just tell me you what you think.

As usual, I'm looking for quality, not quantity but as a ballpark figure I imagine 100-200 words, as reported by `wc -w`, will be needed for a thoughtful answer.

As always, the `.txt` suffix on `twoways.txt` should be enough to tell you that I'm wanting a plain ASCII text file, not a Word document, PDF, etc.

Problem 7. (ZERO points) `pancakes.rb`

Let's see who will write a Ruby version of the Haskell pancake printer from assignment 3 for zero points!

The Ruby version is a program that reads lines from standard input, one order per line, echoes the order, and then shows the pancakes.

Example:

```
$ cat pancakes.1
3 1 / 3 1 5
3 1 3
1 5/          1 1 1/11 3 15     /3 3 3     3/1
1
$ ruby pancakes.rb < pancakes.1
Order: 3 1 / 3 1 5

     ***
***   *
 *  *****

Order: 3 1 3

***
 *
***

Order: 1 5/          1 1 1/11 3 15     /3 3 3     3/1

                        ***
      *   ***********   ***
  *   *       ***       ***
***** * *************** *** *

Order: 1

*

$
```

A blank line is printed after the `Order:` line and again after the stacks.

Assume that input lines consist exclusively of integers, spaces, and slashes, which separate stacks. Assume that there is at least one stack. Assume all stacks have at least one pancake. Assume all widths are greater than zero. Assume the input is well-formed—you won't see something like "`1 / / 3`" or "`/ 3 /`". Assume there are no empty lines in the input.

If an order specifies an even-width pancake, the message shown below is printed. Processing then continues with the next order in the input, if any.

```
$ ruby pancakes.rb < pancakes.2
Order: 1 3 1 / 1 2 3
```

```
Even-width pancake.  Order ignored.

Order: 51 49

************************************************
 ***********************************************

$
```

In case you you want to play "Beat the Teacher" I'll tell you that it took me about 25 minutes to write `pancakes.rb`, sketching on paper included. If you care to, let me know how long it takes you. Think about it all you want to but start the clock the moment a tangible artifact is produced, like a mark on a piece of paper.

Problem 8. Extra Credit `observations.txt`

Submit a plain text file named `observations.txt` with...

(a) (1 point extra credit) An estimate of how long it took you to complete this assignment. To facilitate programmatic extraction of the hours from all submissions have an estimate of hours on a line by itself, more or less like one of the following three examples:

```
Hours: 6
Hours: 3-4.5
Hours: ~8
```

If you want the one-point bonus, be sure to report your hours on a line that starts with `"Hours:"`. Some students are including per-problems times, too. That's useful and interesting data—keep it coming!—but `observations.txt` should have only one line that starts with `Hours:`. If you care to report per-problem times, impress me with a good way to show that data.

Other comments about the assignment are welcome, too. Was it too long, too hard, too detailed? Speak up! I appreciate all feedback, favorable or not.

(b) (1-3 points extra credit) Cite an interesting course-related observation (or observations) that you made while working on the assignment. The observation should have at least a little bit of depth. Think of me saying "Good!" as one point, "Interesting!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use the D2L Dropbox named `a5` to **submit a single zip file named `a5.zip` that contains all your work**. If you submit more than one `a5.zip`, your final submission will be graded. Here's the full list of deliverables:

`nzip.rb`
`nzip.rb`
`mirror.rb`
`calc.rb`
`switched.rb`
`pancakes.rb` *(zero points!)*
`observations.txt` *(for extra credit)*

Note that all characters in the file names are lowercase.

To avoid misdirected submits, the `a5` Dropbox won't appear until after assignment 4 is due.

Miscellaneous

Here's what `wc` shows for my current solutions:

```
$ wc nzip.rb vrepl.rb mirror.rb calc.rb switched.rb pancakes.rb
  11   27  194 nzip.rb
   8   29  140 vrepl.rb
  17   23  205 mirror.rb
  35   72  662 calc.rb
  79  212 1928 switched.rb
  39  112  851 pancakes.rb
 189  475 3980 total
```

Restrictions not withstanding, you can use any elements of Ruby that you desire, but this assignment is written with the intention that it can be completed easily using only the material in this write-up and Ruby slides 1-179.

Point values of problems correspond directly to assignment points in the syllabus. For example, a 10-point problem on this assignment corresponds to 1% of your final grade in the course.

Feel free to use comments to document your code as you see fit, but note that no comments are required, and no points will be awarded for documentation itself. (In other words, no part of your score will be based on documentation.) A # is comment to end of line, unless in a string literal or regular expression. There's no Ruby analog for `/* ... */` in Java and `{- ... -}` in Haskell but you can comment out multiple lines by making them an *embedded document*—lines bracketed with `=begin`/`=end` starting in column 1. RPL 2.1.1 has more on comments.

Remember that late assignments are not accepted and that there are no late days; but if circumstances beyond your control interfere with your work on this assignment, there may be grounds for an extension. See the syllabus for details.

My estimate is that it will take a typical CS junior from 7 to 9 hours to complete this assignment.

Keep in mind the point value of each problem; don't invest an inordinate amount of time in a problem or become incredibly frustrated before you ask for a hint or help. Remember that the purpose of the assignments is to build understanding of the course material by applying it to solve problems. If you reach the seven-hour mark, regardless of whether you have specific questions, it's probably time to touch base with me. Give me a chance to speed you up! **My goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.**

I hate to have to mention it but keep in mind that cheaters don't get a second chance. If you give your code to somebody else and they turn it in, you'll both likely fail the class, and more. (See the syllabus for the details.)