CSC 372, Spring 2016
Assignment 6
Due: Friday, March 25 at 23:59:59

The Usual Stuff

Make an `a6` symlink that references `/cs/www/classes/cs372/spring16/a6`. Test using `a6/tester` (or `a6/t`). Use Ruby 2.2.4.

A note about problems 1-4

Each of the first four problems ask you to write an iterator, which is a single method, rather than a program. The write-up for the first iterator, `eo`, shows a couple of ways to approach edit-run cycle.

Each of the iterators has a "duck typing" specification that describes what the iterator requires of its argument(s), such as being subscriptable or having a `size` or `each` method.

IMPORTANT: The first test for each iterator is a Ruby program with a name like *ITERATOR*`-ap.rb`. The "ap" stands for **all points**—when grading, all the points for the problem will be based on whether you pass the "ap" test. The "ap" tests use dumbed-down classes that provide only the capabilities that the iterator is supposed to require. For example, `eo-ap.rb` uses a class named `Dumb` that provides only subscripting with `x[n]` and a `size` method. If you're passing some cases for an iterator but failing the "ap" test, then your implementation is requiring more capabilities of its argument(s) than it should be, according to the duck typing specification.

Problem 1. (2 points) `eo.rb`

Write a Ruby iterator `eo(x)` that yields every other element in `x`. `eo(x)` returns `x`.

Duck typing: `eo` only requires `x` to be subscriptable with `x[n]` and have a `size` method.

Usage:

```
>> eo([10,20,30,40]) {|v| puts v }
10
30
=> [10, 20, 30, 40]

>> eo("testing") {|c| print "'#{c}' "}
't' 's' 'i' 'g' => "testing"

>> sum = 0; eo((1..10).to_a) {|v| sum += v}; sum
=> 25

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

Here's a way to approach the edit-run cycle with `eo` and the three iterators that follow, `tkcycle`, `vrepl` and `mirror`:

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

```
$eo="eo.rb"
```

After restarting `irb` you can do this:

```
$ irb
>> load $eo
=> true
>> eo("abcd") {|c| puts c}
a
c
=> "abcd"
```

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.

You can edit in another window and then do `load $eo` to load up the latest.

Another angle to save a little typing is add a method like this to to your `.irbrc`:

```
def ld s
    load s.to_s + ".rb"
end
```

Use it like this:

```
$ irb
>> ld :eo
=> true
```

By giving `ld` a symbol, it's sort of like using a string literal that needs a quote on only one end. (*Atoms* in Lisp are similar.) `ld` converts its argument to a string, tacks on the `".rb"` suffix, and calls `load` with the result. Again, we get this sort of flexibility because `load` is a Ruby method, but on the other hand, I still miss the filename completion that `ghci` provides.

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

Write a Ruby iterator `tkcycle(x, sizes)` that yields consecutive "slices" of `x` based on the integers in `sizes`. `tkcycle` cycles through the sizes in `sizes` until it runs out of elements in `x`. `tkcycle` always returns `nil`.

Duck typing: `tkcycle` only requires `x` to support a "slice" operation with `x[start,length]` and have a `size` method. `sizes` is expected to be a non-empty array of integers.

Examples:

```
>> tkcycle((1..10).to_a,[1,2]) {|s| p s}
[1]
[2, 3]
[4]
[5, 6]
[7]
[8, 9]
[10]
=> nil
```

```
>> tkcycle("just a test", [3]) {|s| puts "slice = #{s}"}
slice = jus
slice = t a
slice =  te
=> nil

>> tkcycle("just a test", [30]) {|s| puts "slice = #{s}"}
=> nil

>> tkcycle("just a test", [3,0,2]) {|s| puts ">#{s}<"}
>jus<
><
>t <
>a t<
><
>es<
```

Note that **`tkcycle("just a test", [3])`** above demonstrates that `tkcycle` never yields a partial result: After the third block invocation, with `s = " te"`, only `"st"` remains, and that's not enough for the next three-element yield.

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

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

Duck typing: `vrepl` only requires `x` to have an `each` method.

```
>> 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 }
=> []
```

Problem 4. (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.

Duck typing: `mirror` only requires that `x` implement the iterator `each`.

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 the previous three iterators, `mirror` is a freestanding method.

Problem 5. (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=11111111111111111*11111111111111111
123456790123456787654320987654321
? 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 these 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` or `3*-4`. Division by zero is not supported. There will be no empty lines in the input.

*Implementation note*s

Regular expressions are handy in a couple of places in `calc.rb`. As of press time we're just getting into regular expressions, so I'm going to give a couple of pieces of code to use as-is. The first is a method that can be used to see if a string is a non-negative integer:

```
def isInt(s)
    !!(s =~ /^\d+$/)
end
```

The match simply requires that `s` consist of nothing but digits. I chose to use a couple of "not"s to produce `true` or `false` rather than a truthy match position or a falsy `nil`, mainly for a prettier example below.

The second piece of regular expression code is a particular invocation of `String#scan`, to break up an input line:

```
>> line = "x2=3*val+40-500"

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

Just for fun, let's combine that with `isInt`:

```
>> line.scan(/\w+|\W+/).map {|s| isInt(s)}
=> [false, false, true, false, false, false, true, false, true]
```

WARNING: `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 6. (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 one and all numbers in the rightmost column are less than one.

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 `a6/yob/1951.txt`, which has the 1951 data:

```
$ grep Dana, a6/yob/1951.txt
Dana,F,1076
Dana,M,1277
```

The format of the `a6/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, a6/yob/1958.txt
Dana,F,2388
Dana,M,1531
```

`switched.rb` reads the `a6/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's 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 are 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, a6/yob/yob1949.txt
Lavern,F,93
Lavern,M,121

$ grep Lavern, a6/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. When your `switched.rb` is done, try this:

```
for i in $(seq 1940 2005); 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 2014, and the first must be less than the second.

Implementation notes for `switched`

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/spring16/a6/switched-hint.html

It's easy to drown in the data on a problem like this. You might start by having your code that reads the *YEAR*`.txt` files discard data for everybody but "Dana" and then use "`p x`" (equivalent to "`puts x.inspect`") to dump out your data structure. Then try adding in a male-only and a female-only name, like "Delbert" and "Mimi" in 1951. Alternatively, you might edit down some data files to just a few lines of interest. (After copying any of the `a6/yob` files into your directory, use `chmod 600` *FILE* so that you can edit it; `cp` will have left it as mode `444`—read only.)

Watch out for bugs related to integer division. (Use `.to_f` to get a `Float` when needed.)

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("a6/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 mailing to us or posting on Piazza, take a look at the data files and see if you can answer the question yourself. The files are in `a6/yob`. Those same files will be used for testing when grading.

You can download http://www.cs.arizona.edu/classes/cs372/spring16/a6/yob.zip for testing on your own machine. In the same directory as your `switched.rb`, make a directory named `a6` and then unzip `yob.zip` in that directory to produce a structure compatible with the `File.open` above.

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

Let's see who will write a Ruby version of the Haskell pancake printer from assignment 4 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 a6/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 < a6/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 < a6/pancakes.2
Order: 1 3 1 / 1 2 3

Even-width pancake.  Order ignored.

Order: 51 49

**************************************************
```

```
**************************************************

$
```

In case 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 total (estimated) hours on a line that starts with `"Hours:"`. There must be only one `"Hours:"` line in `observations.txt`. It's fine if you care to provide per-problem times, and that data is useful to us, but report it in some form of your own invention, not with multiple lines that contain "Hours:", in either upper- or lower-case.

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, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use `a6/turnin` to submit your work. Each run creates a time-stamped "tar file" in your current directory with a name like `a`*`N`*`.`*`YYYYMMDD`*`.`*`HHMMSS`*`.tz` You can run `a6/turnin` as often as you want. We'll grade your final submission.

Note that each of the `aN.*.tz` files is essentially a backup, too, but perhaps mail to `372s16` if you need to recover a file and aren't familiar with `tar`—it's easy to accidentally overwrite your latest copies with a poorly specified extraction.

`a6/turnin -l` shows your submissions.

To give you an idea about the size of my solutions, with comments stripped, here's what I see as of press time:

```
$ wc $(grep -v txt a6/delivs)
   8   17   93 eo.rb
  19   37  332 tkcycle.rb
   8   29  158 vrepl.rb
```

```
 17   23  205 mirror.rb
 35   75  667 calc.rb
 57  142 1241 switched.rb
 39  112  851 pancakes.rb
183  435 3547 total
```

Miscellaneous

You can use any elements of Ruby that you desire, but the assignment is written with the intention that it can be completed easily using only the material presented on Ruby slides 1-200.

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.)

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.

Our goal is that everybody gets 100% on this assignment AND gets it done in an amount of time that is reasonable for them.

If you put seven hours into this assignment and don't seem to be close to completing it, it's definitely time to touch base with us, regardless of whether you have any questions. Specifically mention that you've reached seven hours. Give us a chance to speed you up!

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, get a permanent transcript notation stating you cheated, and maybe more. See the syllabus for the details.