CSC 372, Spring 2016
Assignment 7
Due: Friday, April 8 at 23:59:59

Option: Make Your Own Assignment!

If you've got an idea for something you'd like to write in Ruby, you can propose that as a replacement for some or all of the problems on this assignment. To pursue this option send me mail with a brief sketch of your idea. We'll negotiate on points and details.

The Usual Stuff

Make an `a7` symlink that references `/cs/www/classes/cs372/spring16/a7`. Test using `a7/tester` (or `a7/t`). Use Ruby 2.2.4.

Don't hesitate to dig into the test programs

Most of the tests for the problems on this assignment are in the form of Ruby programs that exercise a number of cases. For example, there's only one test for `label.rb`. It's this: `ruby a7/label1.rb`

The tests for `vstring.rb` look like this:

```
ruby a7/vstring1.rb 1r
ruby a7/vstring1.rb 1m
...
```

The command line argument specifies the block of tests to run.

In some cases you may need to dig around in those test programs to figure out exactly what code is being executed for a failing test. In some cases it may be useful to copy the code into your directory and hack it up, maybe adding `Kernel#p` calls or varying the amount of loop iterations to help narrow down a bug.

In various places in the output for the `label` and `VString` tests you'll see something like this:

```
Line 35 in a7/label1.rb:
label([a,b,c]):
   a1:[a2:[],a3:[a2,a2],a4:[a3,a3]]
```

That first line tells us that the test originates at line 35 in `a7/label1.rb`. Here's that line:

```
sv("label([a,b,c])", bb)
```

`sv`, standing for "show values" is a helper method that uses `eval` to evaluate the expression specified by the first argument. The second argument, `bb`, is the current set of variable bindings, which are used by `eval`.

Problem 1. (12 points) `label.rb`

`Array#inspect`, which is used by `Kernel#p` and by `irb`, does not accurately depict an array that contains multiple references to the same array and/or references itself. Example:

```
>> a = []
```

```
>> b = [a,a]

>> p b
[[], []]
```

By simply examining the output of `p b` we can't tell whether `b` references two distinct arrays or has two references to the same array.

Another problem is that if an array references itself, Ruby "punts" and shows "`[...]`":

```
>> a = []

>> a << a

>> p a
[[...]]
```

The problems continue if we then get a `Hash` involved:

```
>> h = {}

>> h[a] = h

>> p h
{[[...]]=>{...}}
```

For this problem you are to write a method `label(x)` that produces a labeled representation of x, where x is a string, number, array or hash. Arrays and hashes may in turn reference other arrays and hashes, and be cyclic.

Here's what `label` shows for the first case above.

```
>> a = []; b = [a,a]

>> puts label(b)
a1:[a2:[],a2]
```

The outermost array is labeled as `a1`. Its first element is an empty array, labeled `a2`. The second element is a reference to that same empty array. Its contents are not shown, only the label `a2`. For reasons described later, we'll use `puts label(...)` to show the string that `label` produces.

Here's another step, and the result:

```
>> c = [b,b]

>> puts label(c)
a1:[a2:[a3:[],a3],a2]
```

Note that the label numbers are not preserved across calls. The array that this call labels as `a3` was labeled as `a2` in the previous example.

To explore relationships between the contents of `a`, `b`, and `c` we could wrap them in an array:

```
>> puts label([a,b,c])
a1:[a2:[],a3:[a2,a2],a4:[a3,a3]]
```

Here's a simple cyclic case. The third element in `a` is a reference to `a`; representing that reference with a label lets us see the cycle.

```
>> a = []

>> a = [10,20]

>> a << a

>> puts label(a)
a1:[10,20,a1]
```

Let's try a simple hash:

```
>> h = {}

>> h["one"] = 1

>> h[2] = "two"

>> puts label(h)
h1:{"one"=>1,2=>"two"}
```

Note that hashes are labeled as "`hN`". The key/value pairs are shown with "thick" arrows. Pairs are separated with commas, and curly braces surround the list of pairs.

Let's add some arrays into the mix:

```
>> h = {}

>> a = [2,2,2]

>> a << a

>> h["twos"] = a

>> h["words"] = %w{just a test}

>> puts label(h)
h1:{"twos"=>a1:[2,2,2,a1],"words"=>a2:["just","a","test"]}
```

Note that arrays and hashes have separate numbering: the above labeling shows both `h1` and `a1`, for example.

Let's try `h[h] = h`:

```
>> h[h] = h

>> puts label(h)
h1:{"twos"=>a1:[2,2,2,a1],"words"=>a2:["just","a","test"],h1=>h1}
```

`label(h)` eventually reaches the key/value pair made by `h[h] = h`. Because `h` has already been labeled as `h1`, the presence of that key/value pair is shown as `h1=>h1`.

Another example:

```
>> a = [1,2,3]

>> a << [[[a,[a]]]]

>> a << a

>> puts label(a)
a1:[1,2,3,a2:[a3:[a4:[a1,a5:[a1]]]],a1]
```

One more example:

```
>> a = [1,2,3]

>> b = {"lets" => "abc", 1 => a}

>> 3.times { a << b }

>> a << "end"

>> puts label([a,b,{100=>200}])
a1:[a2:[1,2,3,h1:{"lets"=>"abc",1=>a2},h1,h1,"end"],h1,h2:{100=>200}]
```

The trivial case for `label(x)` is that `x` is not an array or hash. If so, `label` returns `x.inspect`:

```
>> puts label(4)
4

>> puts label("testing")
"testing"
```

Here are some simple cases that are good for getting started:

```
>> puts label([7])
a1:[7]

>> puts label([[7]])
a1:[a2:[7]]

>> puts label([[70],[80,"90"]])
a1:[a2:[70],a3:[80,"90"]]
```

Keep in mind that your solution must be able to accommodate an arbitrarily complicated structure but the only types you'll encounter are numbers, strings, arrays and hashes.

This routine is a simplified version of the `Image` procedure from the Icon library. I've looked around and asked around for something similar in Ruby. I haven't found anything yet but it may well exist. If you find such a routine, which would trivialize this problem, you may not use it. However, you may study it and then, based on what you've learned, create your own implementation. And, tell me about your discovery!

The above examples use `puts label(...)` rather than showing the result of `label(...)`. Let's look at a `label` call without `puts`:

```
>> label([1,"two",["3"]])
```

```
=> "a1:[1,\"two\",a2:[\"3\"]]"
```

`label` returns a string and `irb` uses `inspect` to show an unambiguous representation of results, so any quotes in the result are escaped. Using `puts` lets us avoid that clutter:

```
>> puts label([1,"two",["3"]])
a1:[1,"two",a2:["3"]]
=> nil
```

Implementation notes

I think of this as a fairly hard problem to solve given only the above and what you've seen in class, but for those who wish to have a challenge, I'll say nothing here about how to approach it. If you have trouble getting started, however, see http://www.cs.arizona.edu/classes/cs372/spring16/a7/label-hint.html

IMPORTANT: You must match the sequence of label numbers that my solution produces. That essentially requires you to traverse the structure in the same order I do, which is depth-first. Here's an example that evidences that depth-first traversal:

```
>> puts label([[[10]],[[21,22]]])
a1:[a2:[a3:[10]],a4:[a5:[21,22]]]
```

Note that the deeply nested `[10]` was labeled with `a3` before the second element of the top-level list was labeled with `a4`.

Also, note that I iterate through key/value pairs in a hash using `h.each {|k,v| ... }`

Problem 2. (15 points, unevenly distributed across four sub-problems) `re.rb`

In this problem you are to write several methods. **Each returns a regular expression that matches the specified strings (no more, no less) and, in some cases, creates named groups.**

Here is an example specification:

> *Write a method `letsdigs_re` that produces a regular expression that matches strings that consist of one or more letters followed by three or more digits. The named groups `lets` and `nums` are set to the letters and digits, respectively. A dash may optionally appear between the two. If so, the named group `dash` is non-empty (i.e., not `nil` and not the empty string).*

Here's a solution: (See slide 231 if you don't recognize the `{3,}` construct.)

```
def letsdigs_re
    /^(?<lets>[a-z]+)(?<dash>-?)(?<digs>\d+{3,})$/
end
```

Because the regular expression is the last expression in `letsdigs_re`, it is the return value.

`spring16/ruby/smg.rb` has a variant of `show_match` (slide 213) that also shows named groups. Maybe add that method to your `~/.irbrc`. Let's test `letsdigs_re` with `smg`:

```
>> smg(letsdigs_re, "abc123")
"<<abc123>>"
lets = <<abc>>, dash = <<>>, digs = <<123>>

>> smg(letsdigs_re, "abc-12")
```

```
no match

>> smg(letsdigs_re, "abc-1234")
"<<abc-1234>>"
lets = <<abc>>, dash = <<->>, digs = <<1234>>
```

A great resource for developing regular expressions is rubular.com. Here's the example above on Rubular: http://rubular.com/r/vPnFMgzPoB. Note that the "Your test string:" window has three test cases, one per line. There is a little bad news: Rubular doesn't seem to provide any way to build up a regular expression like shown on slide 246.

Here's an older resource: a video example of using `show_match` to gradually develop a regular expression: http://screencast.com/t/FO7OOIScCR39. (Yes, I should produce a version of it that uses Rubular!)

The set of tests used for grading `re.rb` will be the set of tests used by `a7/tester`, and that set will be finalized by noon on March 26.

Here are the methods you are to write for this problem:

(a) (1 point) Write a method `phone_re` that produces a regular expression that matches strings that are phone number in any of these forms: `555-1212`, `800-555-1212`, and `(800) 555-1212`.

(b) (2 points) Write a method `sentence_re` that produces a regular expression that matches sentences, as follows: Sentences must begin with a capital letter. Sentences are composed of one or more words. Words are separated by exactly one blank. The sentence must end with a period, question mark, exclamation mark, or one of the two strings `"!?"` and `"?!"`.

Two good sentences: `"I shall test this!"`, `"Xserwr AAA x."`

A bad sentence: `"it works!"` (Doesn't start with a capital.)

See http://www.cs.arizona.edu/classes/cs372/spring16/a7/sentence-hint.html if you have trouble getting started with this problem.

(c) (3 points) Write a method `hours_re` that produces a regular expression that matches a specification of one or more office hours periods, like "`MWF 12:00-13:00`", or "`T-H 9:30-12:30, F 19:00-0:00`". Days are specified with one or more letters in the set `[MTWHF]`, or a range from one day to another. Hours are in the range `0:00-23:45`, with 5-minute resolution. Times before `10:00` can optionally specified with a leading zero, like `09:45`. Multiple periods are separated by a comma and a space.

(d) (3 points) Here are some examples of `ls -l` output:

```
-rw-r-----   1 whm whm    543 Mar 14 18:19 ttl.rb
-rw-r--r--   1 whm whm   6555 Dec 10  2009 w.dat
lrwxrwxrwx   1 whm whm      6 Mar 19 20:56 a7/t -> tester
drwx------ 183 whm empty 1306 Mar 21 20:21 /home/whm/.
drwx--x--x  51 whm wheel  318 Jan 30 22:13 /x/closed
```

The first character indicates file type—`d` for directory, `l` for symbolic link, `-` for regular files. There are other types, too.

The next nine characters, which are three three-character groups, show the permissions. The first group of three characters is "user" permissions—what the owner of the file is permitted to do with the file. The next three characters are the group permissions. The third group of three is "other" permissions—what users who aren't the owner nor are in the appropriate group can do with the file. Ignoring some special cases, the first letter of the three-character groups is always either '`r`' or '`-`'; the second is always '`w`' or '`-`'; and the third is always '`x`' or '`-`'. You'll never see something like '`rrw`' or '`w-w`'.

For this problem you're to write a method `perms_re` that produces a regular expression that matches lines of `ls -l` output for files whose "other" permissions (the third group of three) is not "`---`" **and** the file is not a symbolic link. In the lines above, `w.dat` and `/x/closed` meet that criteria.

When the match is successful, the named group `perms` should contain the three characters of "other" permissions (like "`r--`", "`rwx`", or "`--x`"), and the named group `name` should contain the file name, like "`w.dat`".

Assume that file names do not contain blanks, but you might find it interesting to consider handling that case, too.

Note that the format of the first ten characters never varies for `ls -l` output, but field widths for the rest of the line can vary. The upshot of this is that you can't assume that the file name begins in any particular column.

(e) (3 points) For this problem you're to write a method `vr_re` that matches a string if it is a Vim range specification. A Vim range specification can be a single line specification or two line specifications separated by a comma.

We'll handle the following line specifications

- A simple line number, like `15`
- A dot (`.`) for the current line, or `$` for the last line
- A dot or `$` followed by `+N` or `-N`, where `N` is a positive integer
- A regular expression enclosed in slashes, like `/abc/` or `/^x/`. **Assume** the regular expression doesn't contain any escaped slashes, like in `/a\/b/`.

Here are some valid range specifications:

```
10
10,20
.,20
20,$
/abc/,.
.-5,.+10
/begin/,/end/
```

When a match succeeds, the named groups `from` and `to` should be set, if two line specifications are present. If there's only one, then `$~["to"]` should be `nil` or the empty string.

(f) (3 points) Write a method `path_re` that produces a regular expression that matches UNIX paths and sets the named group `dir` to the directory name, `base` to the filename, minus extension, and `ext` to the extension, which is defined as everything in the filename to the right of the leftmost dot. If an element is not present, then `$~[`*`whatever-group`*`]` should be `nil` or the empty string.

Examples: (excerpt of output from `ruby a7/re1.rb path < a7/re.path`)

```
path: 'longest.java', dir = '', base = 'longest', ext = 'java'

path: '/etc/passwd', dir = '/etc/', base = 'passwd', ext = ''

path: '/cs/www/classes/cs372/spring16/tester/Test.rb',
dir = '/cs/www/classes/cs372/spring16/tester/', base = 'Test',
ext = 'rb'

path: '/home/whm/.bashrc', dir = '/home/whm/', base = '', ext =
'bashrc'
```

The above is skimpy on examples but you'll find plenty in `a7/re.*`. Those are input files for `a7/re1.rb`.

To be clear, `re.rb`, should consist of six methods. It should look something like this:

```
def phone_re
      ...
end

...four more here...

def path_re
    ...
end
```

A note on testing `re.rb`

For testing, "`a7/t re`" will test all the regular expressions but you can use the tester's `-t` option to test just one of the regular expressions. Example:

```
$ a7/t re -t phone
```

Along with "`phone`" there's "`sen`", "`hours`", "`perms`", "`vr`", and "`path`".

Problem 3. (22 points) `vstring.rb`

For this problem you are implement a hierarchy of four Ruby classes: `VString`, `ReplString`, `MirrorString`, and `IspString`. `VString` stands for "virtual string"—**these classes create the illusion of very long strings but relatively little data is needed to create that illusion**.

`VString` serves as an abstract superclass of `ReplString`, `MirrorString`, and `IspString`; it simply provides some methods that are used by those subclasses.

`ReplString` represents strings that consist of zero or more replications of a specified string. Example:

```
$ irb
>> load "vstring.rb"

>> s1 = ReplString.new("abc", 2)
=> ReplString("abc",2)
```

Note that `irb` used `s1.inspect` to produce the string '`ReplString("abc",2)`', that shows the

type, base string, and replication count.

`VString` subclasses support only a handful of operations: `size`, `[n]`, `[n,m]`, `inspect`, `to_s`, and `each`. The semantics of `[n]` and `[n,m]` are the same as for Ruby's `String` class with one exception, described below.

Here are some examples:

```
>> s1.size            => 6

>> s1.to_s            => "abcabc"

>> s1.to_s.class      => String

>> s1[0]              => "a"

>> s1[2,4]            => "cabc"

>> s1[-5,2]           => "bc"

>> s1[-3,10]          => "abc"

>> s1[10]             => nil
```

A `ReplString` can represent a *very* long string:

```
>> s2 = ReplString.new("xy", 1_000_000_000_000)
=> ReplString("xy",1000000000000)

>> s2.size                  => 2000000000000

>> s2[-1]                   => "y"

>> s2[-2,2]                 => "xy"

>> s2[1_000_000_000]        => "x"

>> s2[1_000_000_001]        => "y"

>> s2[1_000_000_001,7]      => "yxyxyxy"
```

Some operations are impractical on very long strings. For example, `s2.to_s` would require a vast amount of memory; but if the user asked for it, we'd let it run. Similarly, `s2[n,m]` has the potential to produce an impractically large result.

A `MirrorString` represents a string concatenated with a reversed copy of itself. Examples:

```
>> s3 = MirrorString.new("1234")       => MirrorString("1234")

>> s3.size                             => 8

>> s3.to_s                             => "12344321"

>> s3[-1]                              => "1"
```

```
>> s3[2,4]                                  => "3443"
```

An `IspString` represents a string with instances of another string interspersed between each character:

```
>> s4 = IspString.new("xyz", "...")    => IspString("xyz","...")

>> s4.to_s                             => "x...y...z"

>> s4[0]                               => "x"

>> s4[1]                               => "."

>> s4[s4.size-1]                       => "z"
```

In the above examples the strings used in the subclass constructors are instances of `String` but they can be an arbitrary nesting of `VString`s, too:

```
>> s5 = IspString.new(MirrorString.new("abc"),
ReplString.new(".",3))
=> IspString(MirrorString("abc"),ReplString(".",3))

>> s5.to_s
=> "a...b...c...c...b...a"

>> s6 = ReplString.new(s5,1_000_000_000_000_000_000)
=>ReplString(IspString(MirrorString("abc"),ReplString(".",3)),100000
0000000000)

>> s6.size
=> 21000000000000000

>> s6[s6.size-20,100]
=> "...b...c...c...b...a"

>> s6[(s5.size)*(1_000_000_000_000_000_000-2),50]
=> "a...b...c...c...b...aa...b...c...c...b...a"
```

To emphasize that the "V" in `VString` stands for "virtual", consider this interaction:

```
>>s=MirrorString.new(ReplString.new("abc",100000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
00))
=>MirrorString(ReplString("abc",10000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000))

>> s.size
=>60000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000

>> s2 = IspString.new(s,s)
=>IspString(MirrorString(ReplString("abc",10000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000000000000000000000000000
```

```
00)),MirrorString(ReplString("abc",10000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000000000)))

>> s2.size.to_s.size
=> 328
```

The iterator `each` is available for `ReplString`, `MirrorString`, and `IspString`. It produces all the characters in turn, as one-character strings. It returns the `VString` it was invoked on.

```
>> s1 = ReplString.new("abc",2)  => ReplString("abc",2)

>> s1.each {|x| puts x}
a
b
c
a
b
c
=> ReplString("abc",2)
```

Be sure to 'include Enumerable' in VString, so that methods like `map`, `sort`, and `all?` work:

```
>> MirrorString.new("abc").map { |c| c }
=> ["a", "b", "c", "c", "b", "a"]

>> MirrorString.new(ReplString.new("abc",2)).sort
=> ["a", "a", "a", "a", "b", "b", "b", "b", "c", "c", "c", "c"]

>> MirrorString.new(
             ReplString.new("a", 100000)).all? { |c| c == "a" }
=> true
```

Using Ranges to subscript VStrings is not supported:

```
>> s1 = MirrorString.new("ab")   => MirrorString("ab")
>> s1[0..-1]
NoMethodError: ...
```

I won't test with ranges.

TL;DR

Here's a quick summary of what's required of the `VString` subclasses and their constructors:

- `ReplString.new(s, n)` where `s` is a non-empty `String` or a `VString` and `n > 0`.
- `MirrorString.new(s)` where `s` is a non-empty `String` or a `VString`.
- `IspString.new(s1,s2)` where `s1` and `s2` are non-empty `String`s or are `VString`s.
- `size`, `inspect`, and `to_s` methods
- Subscripting with `[n]` and `[start,len]`, with the same semantics as for Ruby's `String` class.
- Subscripting with `[Range]` is **not** supported.
- `VString` implements the iterator `each` and `includes Enumerable`.

There is one exception to matching the semantics of `String` when subscripting with `[start,len]`. Note this behavior of `String`:

```
>> s="abc"       => "abc"

>> s[3]          => nil

>> s[3,1]        => ""

>> s[4,1]        => nil
```

You might find it interesting to think about why `String`s have that behavior but we won't match it with `VString`. Instead, if `start` is out of bounds, `nil` is produced:

```
>> s = ReplString.new("abc",2)   => ReplString("abc",2)

>> s[5,10]                       => "c"

>> s[6,10]                       => nil
```

Implementation Notes

***VERY IMPORTANT**: Implement as much functionality as possible in `VString`*

It may help to imagine that there will eventually be dozens of `VString` subclasses instead of just the three specified here. Having that mindset, and wanting to write as little code as possible to implement those dozens of subclasses, should motivate you to **do as much as possible in `VString` and as little as possible in the subclasses**.

My implementations of `ReplString`, `MirrorString`, and `IspString` have only **FOUR** methods: `initialize`, `size`, `inspect`, and `char_at(n)`. All of those methods are tiny—one or two short lines of code (with one exception).

Implementation of subscripting

`VString` itself should have this method:

```
def [](start, len = 1)
    ...
end
```

That defaulted second argument allows this one method to handle both `s[n]` and `s[start,len]`.

Implement this method in terms of calls to `size` and `char_at`, which in turn will resolve to the implementation of those methods in the three subclasses.

With deeply nested constructions of `VString` subclasses there's a potential of producing exponential behavior with subscripting if your `[]` method contains more than one call to `char_at`. Limit your `[]` method to one `char_at` call. (Save the value with `c = char_at(...)` if you need to use it in multiple places.)

Don't get tangled up considering various cases of nesting

Watch out for thinking like this: "What if I've got a `ReplString` that holds a `MirrorString` of a `MirrorString` of a `ReplString` of an `IspString` made of ..."

Instead, just keep in mind that what you can count on is that the values used in the `VString` constructors support `size`, `s[n]`, `s[start,len]`, and `inspect`. Have a duck-typing mindset and write code that uses those operations. (Yes, there's `to_s`, too, but it's problematic with long strings; avoid using it.)

Those double-quotes in `inspect`

Getting the double-quotes right for `inspect` can be a little tricky. Start by getting the following examples working:

```
>> s = ReplString.new("a\nb", 2)
=> ReplString("a\nb",2)

>> s2 = MirrorString.new("x\ty")
=> MirrorString("x\ty")

>> s3 = ReplString.new(s2,10)
=> ReplString(MirrorString("x\ty"),10)
```

When grading, tests will only exercise the `VString` subclasses; `VString` will not be tested directly. (Note that none of the examples above do anything with `VString` itself.)

Saving a little typing when testing

To save myself some typing when testing, I've got these lines at the end of my `vstring.rb`:

```
if !defined? RS
    RS=ReplString
    MS=MirrorString
    IS=IspString
end
```

With those in place, I can do something like this:

```
>> s = IS.new(MS.new(RS.new("abc",2)),"-")
=> IspString(MirrorString(ReplString("abc",2)),"-")
```

Implementing `each`

In retrospect, I believe the slides don't say enough how to put an `each` method in a class like `VString`. I considered dropping it altogether but it's a useful example for `VString` in general, so here is `VString#each`, for you to drop into your `vstring.rb` as-is:

```
def each
    for i in 0...size
        yield self[i]
    end
    self
end
```

Note that `self[i]` will call `VString#[]`, which will in turn use the `char_at` implementation in the subclass for the instance at hand.

Put all four classes in `vstring.rb`

Put the code for all four classes in `vstring.rb`. `VString` needs to be first.

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

Here's something I saw in a book:

```
class Fixnum
    def hours; self*60 end  # 60 minutes in an hour
end

>> 2.hours       => 120

>> 24.hours      => 1440
```

You are to write a Ruby method `gf(spec)` that dynamically adds (i.e., "monkey patches") a number of such methods into `Fixnum`, as directed by `spec`. Example:

```
gf("foot/feet=1,yard(s)=3,mile(s)=5280")
```

Using `Kernel#eval`, the above call to `gf` adds nine methods to `Fixnum`. Here are six of them: `foot`, `feet`, `yard`, `yards`, `mile`, `miles`. Respectively, on a pair-wise basis, those methods produce the `Fixnum` (which is `self`) multiplied by `1`, `3`, and `5280`.

```
% irb -r ./gf.rb
>> gf("foot/feet=1,yard(s)=3,mile(s)=5280")       => true

>> 1.foot          => 1

>> 10.feet         => 10

>> 5.yards         => 15

>> 3.miles         => 15840

>> 8.mile          => 42240

>> 1.feet          => 1
```

It would perhaps be useful to detect plurality mismatches like `8.mile` and `1.feet` and produce an error but that is not done.

In addition to the six methods mentioned above, these three are added to `Fixnum`, too: `in_feet`, `in_yards`, and `in_miles`:

```
>> (30.feet+10.yards).in_yards   => 20.0

>> 10_000.feet.in_miles          => 1.89393939393939
```

Two more examples:

```
>> gf("second(s)=1,minute(s)=60,hour(s)=3600,day(s)=86400")=> true

>> (12.hours+30.minutes).in_days       => 0.520833333333333

>> gf("inch(es)=1,foot/feet=12")       => true
```

```
>> 18.inches.in_feet                    => 1.5

>> 1.foot                               => 12

>> 1.foot.in_inches                     => 12.0
```

Note that methods later generated by `gf` simply replace earlier methods of the same name. After the two calls `gf("foot/feet=1")` and `gf("foot/feet=12")`, `1.foot` is `12`.

An individual mapping must be in the form *`singular/plural=integer`* or *`singular(pluralSuffix)=integer`*. None of the parts may be empty. Mappings are separated by commas. Only lowercase letters are permitted in the names. No whitespace is allowed. If any part of a specification is invalid, a message is printed and `false` is returned; but the result is otherwise undefined. Here is an example of the output in the case of an error:

```
>> gf("foot/feet=1,yards=3")
bad spec: 'foot/feet=1,yards=3'
=> false
```

Note that the error is not pin-pointed—the specification as a whole is cited as being invalid.

Here are more examples of errors:

```
gf("foot/feet=1,")           # trailing comma
gf("foot/feet=1.5")          # non-integer
gf("foot/=1")                # empty plural
gf("inch()=12")              # empty plural suffix
gf("foot/feet=1,Yard(s)=3") # capital letter
```

This is NOT a restriction but to get more practice with regular expressions I recommend that your solution not use any string comparisons; use matches (=~) to break up the specification. And, using regular expressions will probably increase the likelihood that you accept exactly what's valid.

For this problem you are to use `eval` (slide 261+) to add the methods to `Fixnum`, but as the slides show, using `eval` to generate code based on data supplied by another person can be perilous! `eval` is a powerful tool but you've got to be careful when using it. Googling for `ruby eval` turns a lot of discussion about it, but be wary of those who say, "Never use `eval`!" Instead, recognize that `eval` is a powerful tool and use it with caution, weighing risks and benefits, and **ALWAYS** being careful to consider the source of all data that can possibly contribute directly or indirectly to a string that is passed to `eval`.

Keep in mind that you're writing a method named `gf`, not a program. Helper methods are permitted.

In assignment 3's write-up for `editstr` it was mentioned that the bindings for `x`, `len`, etc. are the beginnings of a simple internal DSL (Domain Specific Language). The list `[x 2, len, x 3, rev, xlt "1" "x"]` uses the facilities of Haskell to specify computation in a new language that's specialized for string manipulation. Similarly, this problem provides another example of an internal DSL by using the facilities of Ruby to express computations involving unit conversions.

Problem 5. (24 points) `optab.rb`

When writing this problem one of my favorite quotes came to mind:

> "Far better is it to dare mighty things, to win glorious triumphs, even though checkered by failure, than to take rank with those poor spirits who neither enjoy much nor suffer much because they live in the gray twilight that knows neither victory nor defeat."—Theodore Roosevelt

One way to learn about a language is to manually create tables that show what type results from applying a binary operator to various pairs of types. For this problem you are to write a Ruby program, `optab.rb`, that generates such tables for Java, Ruby, and Haskell.

Here's a run of `optab.rb`:

```
% ruby optab.rb ruby "*" ISA
 * | I  S  A
---+---------
 I | I  *  *
 S | S  *  *
 A | A  S  *
```

The first argument, `ruby`, specifies Ruby as the language of interest for this run.

The second argument, `"*"`, specifies the operator of interest. We'll make a practice of putting quotes around the operator because some operators, like `*` and <, are shell metacharacters. `\*` would work, too.

The third argument, `ISA`, specifies types of interest. The letters `I`, `S`, and `A` stand for `Fixnum` (`I` for integer), `String`, and `Array`, respectively.

`optab`'s output is a table showing the type that results from applying the operator to various pairs of types. The row headings on the left specify the type of the left-hand operand. The column headings along the top specify the type of the right-hand operand.

The upper-left entry, an `I`, shows that `Fixnum * Fixnum` produces a `Fixnum`. (Remember that we're using `I`, not `F`, to stand for integers.) The lower-left entry, `A`, shows that `Array * Fixnum` produces an `Array`. The `S` in the bottom of the middle row shows that `Array * String` produces a `String`.

The `*`'s indicate that `Fixnum * String`, `String * String`, and three other type combinations produce an error.

Here's an example with Java:

```
% ruby optab.rb java "*" IFDCS
 * | I  F  D  C  S
---+---------------
 I | I  F  D  I  *
 F | F  F  D  F  *
 D | D  D  D  D  *
 C | I  F  D  I  *
 S | *  *  *  *  *
```

`I`, `F`, `D`, `C`, and `S` stand for `int`, `float`, `double`, `char`, and `String`, respectively.

Here's how `optab` is intended to work:

> For the specified operator and types, try each pairwise combination of types with the operator by executing that expression in the specified language and seeing what type is produced, or if an error is produced. Collect the results and present them in a table.

The table just above was produced by generating and then running each of twenty-five different Java programs and analyzing their output. Here's what the first one looked like:

```
% cat checkop.java
public class checkop {
    public static void main(String args[]) {
        f(1 * 1);
        }
    private static void f(Object o) {
        System.out.println(o.getClass().getName());
        }
    }
```

Note the third line, `f(1 * 1);` That's an `int` times an `int` because the first operation to test is `I * I`.

Remember: Ruby code wrote that Java program!

In Ruby, the expression `` `some-command-line` `` is called *command expansion*. It causes the shell to execute that command line. The complete output of the command is collected, turned into a string, possibly with many newlines, and is the result of `` `...` ``. (Note that ` is a "back quote".)

Here's a demonstration of using `` `...` `` to compile and execute `checkop.java`:

```
>> result = `bash -c "javac checkop.java && java checkop" 2>&1`
=> "java.lang.Integer\n"
```

The extra stuff with `bash -c ... 2>&1` is to cause error output, if any, to be collected too.

Here's the `checkop.java` that's generated for `I * S`:

```
public class checkop {
    public static void main(String args[]) {
        f(1 * "abc");
        }
    private static void f(Object o) {
        System.out.println(o.getClass().getName());
        }
    }
```

Note that it is identical to the `checkop.java` generated for `I * I` with one exception: the third line is different: instead of being `1 * 1` it's `1 * "abc"`.

Let's try compiling and running the `checkop.java` just above, generated for `I * S`:

```
% irb
>> result = `bash -c "javac checkop.java && java checkop" 2>&1`
=> "checkop.java:3: error: bad operand types for binary operator
'*'\n                    f(1 * \"abc\");\n
^\n  first type:  int\n  second type: String\n1 error\n"
```

`javac` detects incompatible types for `*` in this case and notes the error. `java checkop` is not executed because the shell conjunction operator, `&&`, requires that its left operand (the first command) succeed in order for execution to proceed with its right operand (the second command).

That output, "`checkop.java:3: ...`" can be analyzed to determine that there was a failure. Then, code maps that failure into a "`*`" entry in the table.

Let's try Haskell with the `/` operator. "`D`" is for `Double`.

```
% ruby optab.rb haskell "/" IDS
 / | I  D  S
---+---------
 I | *  *  *
 D | *  D  *
 S | *  *  *
```

For the first case, `I / I`, Ruby generated this file, `checkop.hs`:

```
% cat checkop.hs
(1::Integer) / (1::Integer)
:type it
```

Note that just a plain `1` was good enough for Java since the literal `1` has the type `int` but with Haskell we use `(1::Integer)` to be sure the type is `Integer`. (Yes; `Integer`, not `Int`.)

Let's try running it. For Java we used `javac` and `java`. We'll use `ghci` for Haskell and redirect from `checkop.hs`. Be sure to specify the **`-ignore-dot-ghci`** option, too!

```
% irb
>> result = `bash -c "ghci -ignore-dot-ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for
help\n[lots more]... linking ... done.\nLoading package base ...
linking ... done.\nPrelude> \n<interactive>:2:14:\n    No instance
for (Fractional Integer)\n      arising from a use of `/'\n[lots
more]\n"
```

Ouch—an error! That's going to be a "`*`".

Here's the `checkop.hs` file generated for `D * D`:

```
% cat checkop.hs
(1.0::Double) * (1.0::Double)
:type it
```

Let's try it:

```
>> result = `bash -c "ghci < checkop.hs" 2>&1`
>> result = `bash -c "ghci -ignore-dot-ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.4.1: http://www.haskell.org/ghc/  :? for
help\nLoading package ghc-prim ... linking ... done.\nLoading
package integer-gmp ... linking ... done.\nLoading package base ...
linking ... done.\nPrelude> 1.0\nPrelude> it :: Double\nPrelude>
Leaving GHCi.\n"
```

If we look close we see `it`, with a type: **`it :: Double`**

In pseudo-code, here's what `optab` needs to do:

For each pairwise combinations of types specified on the command line...

Generate a file in the appropriate language to test the combination at hand.

Run the file using command expansion (`` `...` ``).

Analyze the command expansion result, determining either the type produced or that an error was produced.

Add an appropriate entry for the combination to the table—either a single letter for the type or an asterisk to indicate an error.

The examples above show Java and Haskell testing programs and their execution. You'll need to figure out how to do the same for Ruby, but let us know if you have trouble with that. The obvious route with Ruby is creating and running a file but you can use `Kernel#eval` instead. If you take the `eval` route, you'll probably need to do a bit of reading and figure out how to catch a Ruby exception using `rescue`.

I chose the names `checkop.java` and `checkop.hs` but you can use any names you want.

Below is an example of a complete program that generates a file named `hello.java` and runs it. Note that the program's command-line argument is interpolated into the "here document", which is a multi-line string literal. (See slide 79.)

```
% cat a7/mkfile.rb
prog = <<X
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, #{ARGV[0]}!");
        }
    }
X
# IMPORTANT: That X just above MUST BE IN THE FIRST COLUMN!

f = File.new("hello.java","w")
f.write(prog)
f.close
result = `bash -c "javac hello.java && java hello" 2>&1`
puts "Program output: (#{result.size} bytes)", result

% ruby mkfile.rb whm
Program output: (12 bytes)
Hello, whm!
```

Here's the file that was created:

```
% cat hello.java
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, whm!");
        }
    }
```

Copy `a7/mkfile.rb` into your `a7` directory on lectura and try it, to help you get the idea of generating a program, running it, and then doing something with its output.

If you like to be tidy, you can use `File`'s `delete` class method to delete `hello.java`: `File.delete("hello.java")`

Here's a table that shows what types must be supported in each language, and a good expression to use for testing with that type.

Letter	Haskell	Java	Ruby
`I`	`(1::Integer)`	`1`	`1`
`F`	`(1.0::Float)`	`1.0F`	`1.0`
`D`	`(1.0::Double)`	`1.0`	not supported
`B`	`True`	`true`	`true`
`C`	`'c'`	`'c'`	not supported
`S`	`"abc"`	`"abc"`	`"abc"`
`O`	not supported	`new Object()`	not supported
`A`	not supported	not supported	`[1]`

`optab.rb` is not required to do any error checking at all. It assumes the first argument is `haskell`, `java`, or `ruby`. It assumes the second argument is a valid operator in the language specified. It assumes the third argument is a string of single-letter type specifications and that those types are supported for the language at hand. Behavior is undefined for all other cases. I won't test any error cases.

I hope that everybody recognizes that there needs to be language-specific code for running the Java, Haskell, and Ruby tests but ONE body of code can be used to process command-line arguments, launch the language-specific tests, and build the result table.

Think in terms of an object-oriented solution, perhaps with an `OpTable` class that handles the language-independent elements of the problem. `OpTable` would have subclasses `JavaOpTable`, `HaskellOpTable`, and `RubyOpTable` with language-specific code.

For example, my solution has a method `OpTable#make_table` that handles table generation. It calls a subclass method `tryop(op, lhs, rhs)` to try an operator with a pair of operands and report what's produced (a type or an error). With Java a call might be `tryop("+", "1", '"abc"');` it would return `"S"`. In contrast, `RubyOpTable#tryop("+", "1", '"abc"')` produces `"*"`.

Note that testing the Java cases can be slow. With my version, `ruby optab.rb java "*" IFDCS` takes almost 30 seconds to run on lectura. The same test for Haskell takes about seven seconds.

Ruby's command expansion (`` `...` ``) works on Windows but I haven't tried to work out command lines that'll behave as well as the examples above, which were done on lectura. The bottom line is that you'll probably need to do much of your testing on lectura. However, if you use an `eval`-based approach for Ruby, you can easily get that working on Windows. If you want to write code that runs on both Windows and UNIX, you can use `RUBY_PLATFORM` as a simple way to see what sort of system you're

running on.

For three points of extra credit per language, have your `optab.rb` support up to three additional languages of your choice. PHP, Python, and Perl come to mind as easy possibilities. (For Python, you can do either Python 2 or Python 3, but not both for credit.) At least three types must be supported for each language. You may introduce types in addition to those shown above. Submit a **plain text file** `optab.txt`, that shows your extended version in action. Demonstrate at least three operators for each language. The burden of proof for this extra credit is on you, not me!

Problem 6. Extra Credit `vstring-extra.txt`

For three points of extra credit, devise and implement another subclass for `VString`. For example, a fourth subclass I considered having you implement for `VString` was to be created like this:

```
XString.new("a","bb", 10)
```

It represents the following sequence of characters, which ends with 10 "a"s and 20 "b"s:

```
abbaabbaaabbb...aaaaaaaaaabbbbbbbbbbbbbbbbbb
```

You can't implement `XString` for credit but I'll be impressed if you do it for fun.

Add whatever new `VString` subclass you come up with to your `vstring.rb` and create `vstring-extra.txt`, a plain text file that talks about your creation and shows it action with `irb`.

Turning in your work

Use `a7/turnin` to submit your work.

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

```
$ wc $(grep -v txt a7/delivs)
  21   53  573 label.rb
  29   47  599 re.rb
 104  216 1821 vstring.rb
  35   88  870 gf.rb
 148  328 3440 optab.rb
 337  732 7303 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 in the full set of Ruby slides.

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 10 to 15 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 ten 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 ten 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.