CSC 372, Spring 2015
Assignment 6
Due: Thursday, April 9 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.

Use Ruby 1.9.3!

Use Ruby 1.9.3 for this assignment. See the assignment 5 write-up for how-to details.

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 `a6` and `t` in your assignment 6 directory on lectura, for easy access to the tester and the data files. (And, the tester assumes the `a6` symlink is present.) See the assignment 5 write-up for how-to details.

Overlap Warning!

There may be a very small Prolog assignment that overlaps with this assignment, perhaps even being due before this assignment! It'll be a bit like the `eca1` and `eca2` assignments, but it won't be extra credit.

Problem 1. (25 points) `vstring.rb`

For this problem you are implement a hierarchy of three Ruby classes: `VString`, `ReplString`, and `MirrorString`. `VString` stands for "virtual string"—**these three 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` and `MirrorString`; it simply provides some methods that are used by both `ReplString` and `MirrorString`.

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

```
% irb -r ./vstring.rb
>> s1 = ReplString.new("abc", 2)      => ReplString("abc",2)
```

Note that it is `inspect` that is producing the string '`ReplString("abc",2)`', which shows the type, base string, and replication count. (`irb` uses `inspect` to show the result of each expression.)

A handful of operations are supported: `size`, `[n]`, `[n,m]`, `inspect` (used to show results in `irb`), `to_s`, and `each`. The semantics of `[n]` and `[n,m]` are the same as for Ruby's `String` class. 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"
```

A `ReplString` or a `MirrorString` can be made from a `ReplString` or a `MirrorString`. Here is a simple example, a string made of three replications of two replications of `"123"`:

```
>> s1 = ReplString.new(ReplString.new("123",2),3)
=> ReplString(ReplString("123",2),3)

>> s1.to_s      => "123123123123123123"
```

Below is a `ReplString` inside a `MirrorString` inside a `ReplString`. I've added a method called `commas` to `Integer` (a superclass of `Fixnum` and `Bignum`) that displays the values with commas added for readability. (To use it, add `require "a6/commas"` at the top of your `vstring.rb`.)

```
>> s4 = ReplString.new("0123456789", 1_000_000_000)
=> ReplString("0123456789",1000000000)

>> s4.size.commas                => "10,000,000,000"

>> s5 = MirrorString.new(s4)
=> MirrorString(ReplString("0123456789",1000000000))


>> s5[10_000_000_000-1,2]        => "99"

>> s5.size.commas                => "20,000,000,000"

>> s6 = ReplString.new(s5, 1000) =>
ReplString(MirrorString(ReplString("0123456789",1000000000)),1000)

>> s6.size.commas                => "20,000,000,000,000"

>> s6[20_000_000_000-2,4]        => "1001"
```

Here's an example that hopefully emphasizes the "V" for "virtual" in `VString`:

```
>>s=MirrorString.new(ReplString.new("abc",10000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
00000))
=>MirrorString(ReplString("abc",10000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000000000000000000000000000000000000000000))

>> s.size.commas
=>"60,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,0
00,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,
000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000,000
,000,000,000,000,000"
```

The iterator `each` is available for both `ReplString` and `MirrorString`. 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)

>> MirrorString.new("abc").each { |x| puts x }
a
```

```
b
c
c
b
a
=> MirrorString("abc")
```

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
```

Assume that the base string for `ReplString` and `MirrorString` is not empty and that the `ReplString` replication count is greater than zero. In practical terms this means that the shortest possible `ReplString` has a size of 1 and the shortest possible `MirrorString` has a size of 2.

Using `Range`s to subscript `VString`s 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:

- `ReplString.new(s, n)` where `s` is a non-empty `String` or `VString` and `n > 0`.
- `MirrorString.new(s)` where `s` is a `String` or `VString`.
- `size`, `inspect`, and `to_s` methods
- Subscripting with `[n]` and `[n,m]`, with the same semantics as for Ruby's `String` class.
- Subscripting with `[`*`Range`*`]` is **not** supported.
- The iterator `each` works with `ReplString`s and `MirrorString`s.
- `Enumerable` is `included`.

Implementation Notes

SUPER IMPORTANT:
Implement as much functionality as possible in `VString`.

My implementations of `ReplString` and `MirrorString` have only **FOUR** methods: `initialize`, `size`, `inspect`, and `char_at(n)`. All of those methods are tiny—one or two short lines of code. It may help to imagine that there will eventually be dozens of `VString` subclasses instead of just `ReplString` and `MirrorString`. 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.

When grading, tests will rely only on `ReplString` and `MirrorString`; `VString` will not be tested directly. (Note that none of the examples above do anything with `VString`.)

One way to save some typing when doing manual testing is to initialize `RS` and `MS` with `ReplString` and `MirrorString`. I've got these lines at the end of my `vstring.rb`:

```
MS=MirrorString
RS=ReplString
```

With those in place, I can do this:

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

Problem 2. (14 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`, that call to `gf` adds six methods to `Fixnum`: `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, three others are added, 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 241+) to add the methods to `Fixnum`, but note that in general 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`!" (They might also say to never drive on the wrong side of the road; but if you don't, you might get stuck behind a lot of tractors when you're out in the country!)

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

In assignment 2'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 3. (6 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 to 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":

```
>> a = []

>> a << a

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

For this problem you are to write a method `label(a)` that produces a labeled representation of an array that may reference other arrays and/or contains strings and/or numbers. 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 with `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`. 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]]
```

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 = ["abc"]

>> 3.times { a << b; b << a }

>> puts label([a,b])
a1:[a2:[1,2,3,a3:["abc",a2,a2,a2],a3,a3],a3]
```

Some simple cases:

```
>> 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 array. However, the only types you'll encounter are numbers, strings, and arrays. You won't see something like a hash that contains arrays of hashes with arrays for both keys and values, for example.

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!

Implementation notes

My solution is recursive. It has nine lines of code and starts like this:

```
def label(x, h = {})
    return x.inspect if !x.is_a? Array
```

Use `Array#object_id` to produce a unique integer id for each array. Perhaps use that id as a key in a

hash, with the value being a label, like `"a1"`.

You must match my sequence of labels. That essentially requires you to traverse the structure in the same order I do, which is depth-first. Here's an example that illustrates that:

```
>> 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`.

Problem 4. (14 points, unevenly distributed across four sub-problems) `re.rb`

In this problem you are to write four 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 `oddnum_re` that produces a regular expression that matches strings that represent an odd integer.

Here is the solution:

```
def oddnum_re
    /^-?\d*[13579]$/
end
```

I recommend using `show_match` from slide 192 with `irb` to develop regular expressions. Here's a video example of using `show_match` to gradually build up a desired regular expression: http://screencast.com/t/FO7OOlScCR39.

I've got `show_match` in my `~/.irbrc`. If you don't already have a `~/.irbrc` on lectura you can get a copy of one that I've made by doing this:

```
$ cp a6/.irbrc ~
```

Slides 216-220 talk about using groups and referencing them with `\n` or `$n` but it can be tedious to make a group have particular number, like 3 or 5, so `range_re` and `path_re` require you to use *named groups* instead of numeric group references. As an example of using named groups, here is `group1.rb` from slide 219 rewritten to use named groups:

```
while line = gets
    if line =~ /((?<op1>\d+)\+(?<op2>\d+))/ then
        left = $~["op1"].to_i
        right = $~["op2"].to_i

        puts "#{$1} = #{left + right}"
    else
        puts "?"
    end
end # group1_named.rb
```

First, notice the regular expression:

```
/((?<op1>\d+)\+(?<op2>\d+))/
```

The strings `?<op1>` and `?<op2>` specify group names by virtue of appearing immediately after the opening parentheses of their respective groups.

Next, notice these two lines:

```
left = $~["op1"].to_i
right = $~["op2"].to_i
```

The predefined global variable `$~` is an instance of the `MatchData` class. After a successful match it encapsulates all the data about the match. `$~[`*`gname`*`]` is the string matched by the group whose name is *`gname`*. Thus, `$~["op1"]` is the string of digits matched by group that starts with `?<op1>`.

Named groups are also called "named captures".

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

(a) (2 points) Write a method `range_re` that produces a regular expression that matches strings that represent a Ruby `Range` with integer literals, like `1..10` and `-20...10`. The named group `from` is set to the first number, the named group `to` is set to the second number; and if the named group `tdr` is not empty, it indicates a three-dot range was matched.

(b) (3 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.)

For this method, `sentence_re`, here's a simple progression you might follow to develop a solution:

1. Match sentences of the form "a.", "aaaa.", "aa." -- just one word that's one more or "a"s followed by a period.

2. Match sentences of the form "a.", "a a.", "a a a a a." -- one or more words separated by one space but all words are "a".

3. Match "a.", "aaaa a aaa.", "a a a aaaaaaaaa."

4. Like previous but first letter must be capitalized, like "A.", "Aa a aaaa.", etc.

5. Like previous but also handle the endings: "A a!", "Aaaa a aa?!", etc.

6. Finally, allow words to composed of letters other than just 'a's.

(c) (4 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, the group is set to the empty string.

Examples: (excerpt of output from `ruby a6/re1.rb < a6/re.path`)

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

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

path: '/cs/www/classes/cs372/spring15/tester/Test.rb',
dir = '/cs/www/classes/cs372/spring15/tester/', base = 'Test',
ext = 'rb'

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

(d) (5 points) Write a method `calc_re` that matches valid input lines for `calc.rb`, from assignment 5.

A particular hazard with `calc_re` is that you can create a repetitious mess that you won't be able to understand or debug. I recommend creating regular expressions to represent components of `calc` input lines and then creating a regular expression that uses those components to describe a full line. Here's an example of the technique:

```
def fpexp_re
  number = /(\d*\.\d+|\d+)/
  op = /[-+*\/]/
  /^#{number}(#{op}#{number})*$/
end
```

For this example but not `calc_re`, numbers can be in one of three forms: `100`, `10.01`, `.123`. A regular expression that simply recognizes numbers of those forms is assigned to `number`. A regular expression that recognizes a single-character operator is assigned to `op`. The final result of `fpexp_re` is a regular expression that matches `number` followed by zero or more repetitions of `op` followed by `number`. Anchors are used to match the full line.

Usage: (My `.irbrc` has an `sm` alias for `show_match`.)

```
>> sm(fpexp_re, "2.3")
=> "<<2.3>>"

>> sm(fpexp_re, "2.3/10*.7")
=> "<<2.3/10*.7>>"

>> sm(fpexp_re, "20.+10")          ("20." is not allowed)
=> "no match"
```

Here's the harder-to-read alternative that's just one big regular expression:

```
def fpexp_re
  /^(\d*\.\d+|\d+)([-+*\/](\d*\.\d+|\d+))*$/
end
```

See http://www.cs.arizona.edu/classes/cs372/spring15/a6/calc-hint.html if you need a hint on `re_calc`.

The above is skimpy on examples but you'll find plenty in `a6/re.1`. That's an input file for `a6/re1.rb`. It includes the `oddnum_re` example mentioned above.

To be clear, `re.rb`, should consist of four methods: `range_re`, `sentence_re`, `path_re`, and `calc_re`. It should look something like this:

```
def range_re
     ...
end

def sentence_re
    ...
end

def path_re
    ...
end

def calc_re
    ...
end
```

A note on testing `re.rb`

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

```
$ t re -t r
...
Test: 'ruby /cs/www/classes/cs372/spring15/a6/re1.rb <
/cs/www/classes/cs372/spring15/a6/re.range': PASSED
```

That "`r`" stands for range. There's also "`s`", "`p`", and "`c`".

Problem 5. (26 points) `optab.rb`

When writing this problem this quote 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 `` `...` ``.

Here's a demonstration of using `` `...` `` to compile and execute `checkop.java`, which Ruby code generated.

```
>> 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 was 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` produced for `I * I` with one exception: the third line is different: instead of being `1 + 1` it's `1 * "abc"`.

Let's try command expansion with the `checkop.java` just above, for `I * S`:

```
% irb
>> result = `bash -c "javac checkop.java && java checkop" 2>&1`
=> "checkop.java:3: operator * cannot be applied to
int,java.lang.String\n        f(1 * \"abc\");\n             ^\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`:

```
% irb
>> result = `bash -c "ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for
help\n[lots more]... linking ... done.\n> \n<interactive>:2:14:\n
No instance for (Fractional Integer) 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
(2.0::Double) / (2.0::Double)
:type it
```

Let's try it:

```
>> result = `bash -c "ghci < checkop.hs" 2>&1`
=> "GHCi, version 7.6.3: http://www.haskell.org/ghc/  :? for
help\nLoading package ghc-prim ... linking ... done.\nLoading
package integer-gmp ... linking ... done.\nLoading package base ...
linking ... done.\n> 1.0\n> it :: Double\n> 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 me 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 73.)

```
% cat mkfile.rb
prog = <<X
public class hello {
    public static void main(String args[]) {
        System.out.println("Hello, #{ARGV[0]}!");
        }
    }
X
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!");
        }
    }
```

`mkfile.rb` is in `a6`. Copy it into your directory on lectura (`cp a6/mkfile.rb .`) 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.

Maybe think in terms of an object-oriented solution, with a `Language` class that handles the language-independent elements of the problem. `Language` would have subclasses `Java`, `Haskell`, and `Ruby` with language-specific code.

For example, my solution has a method `Language#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, `Ruby#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 `labelplus.txt`

For up to five points make your `label.rb` able to handle hashes, too. Submit a plain text file named `labelplus.txt` showing your enhanced `label(x)` method in operation. To earn all five points your examples will need to convince me that your enhanced method is completely general.

Problem 7. 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, "Excellent!" as two points, and "Wow!" as three points. I'm looking for quality, not quantity.

Turning in your work

Use the D2L Dropbox named `a6` to **submit a single zip file named `a6.zip` that contains all your work**. If you submit more than one `a6.zip`, your final submission will be graded. Here's the full list of deliverables:

`vstring.rb`
`gf.rb`
`label.rb`
`re.rb`
`optab.rb`
`observations.txt` (for extra credit)
`optab.txt` (for extra credit)

Note that all characters in the file names are lowercase.

Miscellaneous

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

```
$ wc vstring.rb gf.rb label.rb re.rb optab.rb
  81  169 1450 vstring.rb
  38   99  833 gf.rb
   9   38  239 label.rb
  17   23  341 re.rb
 133  323 3264 optab.rb
 278  652 6127 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 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.) 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 10 to 15 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 ten-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.)