University of Arizona, Department of Computer Science

CSc 372 — Assignment 7 — Due noon, Wed Nov 23 — 5%

Christian Collberg

November 10, 2011

1 Introduction

The purpose of this assignment is to get started writing Ruby programs.

1. We will be grading on correctness and style. Every procedure should be adequately formatted and documented.
2. Note that the code we're writing in this assignment will be used in the next!
3. The answers to the problems in the first few sections should be put in the file `ass.rb`. The answers for Sections 6 and 7 should be put in the file `fb.rb`.
4. The first few sections should be done *individually*. Sections 6 and 7 can be done in a team of two students.
5. You can read about *here-documents* here: `http://ruby-doc.org/docs/ProgrammingRuby/html/language.html`.

2 Ruby Strings

1. Write a method `calcForm(left, op, right, result)` which uses *here-documents* and *string interpolation* to generate HTML for an on-line calculator. The following code should be returned, with `?????` replaced with the input arguments:

```
<input type="text" name="left" value="?????">
<select name="op">
   <option value="add" ?????>+</option>
   <option value="mul" ?????>*</option>
</select>
<input type="text" name="right" value="?????">
=
?????
```

For example, `calcForm(6, "mul", 7, 42)` should return the string [5 points]

```
<input type="text" name="left" value="6">
<select name="op">
   <option value="add">+</option>
   <option value="mul" selected>*</option>
</select>
```

```
<input type="text" name="right" value="7">
=
42
```

and `calcForm(99, "add", 1, 100)` should return

```
<input type="text" name="left" value="99">
<select name="op">
   <option value="add" selected>+</option>
   <option value="mul">*</option>
</select>
<input type="text" name="right" value="1">
=
100
```

In other words, the word `selected` should appear after `"add"` or `"mul"` depending on the value of `op`, the values of `left` and `right` should be filled in in `value="..."`, and the `result` should appear on the last line.

3 Regular expressions

The problems in this section should be solved using regular expressions.

1. A username consists of a letter followed by at a least one (but possibly more) letters and digits. A password consists of four or more letters or digits. A gender specification consists of either the word `"Male"` or the word `"Female"`. Finish the following predicates: [5 points]

```
def okUsername(un)
   if un =~ /YOUR CODE/ then true else false end
end

def okPassword(pw)
   if pw =~ /YOUR CODE/ then true else false end
end

def okSex(sex)
   if sex =~ /YOUR CODE/ then true else false end
end
```

For example, this code

```
print okUsername("lonelygirl15"), " ", okUsername("15lonelygirl"), "\n"
print okPassword("foo15"), " ", okPassword("foo"), " ", okPassword("#####"), "\n"
print okSex("Male"), " ", okSex("Female"), " ", okSex("male"), " ", okSex("blah"), "\n"
```

should produce this result:

```
true false
true false false
true true false false
```

2. The following method reads and returns lines from a file. Modify it so that it ignores (does not return) any line consisting only of whitespace (blanks and tabs) and any line whose first character is a `#`: [5 points]

```
def read_line (database)
   File.open(database) do |file|
      file.each do |line|
         yield line
      end
   end
end
```

For example, this code

```
read_line("testfile") do |line|
   puts line
end
```

given this input file

```
line 1

line 2

line 3

#don't print this line

  # line 4
```

should yield this output:

```
line 1
line 2
line 3
  # line 4
```

3. Modify this method so that it splits up a string in words and returns it as an array. A word is defined as a sequence of characters surrounded by whitespace, *or* enclosed between double-quotes. You can assume that there is an even number of double-quotes. [5 points]

```
def split_line (line)
   a = []
   line.scan(/YOUR CODE HERE/) do |word|
      a << YOUR CODE HERE
   end
   return a
end
```

For example, this code

```
a = split_line('  aaa "bbb ccc" ddd   "eee"')
a.each do |word|
   puts "'" + word + "'"
end
```

should produce this output:

```
'aaa'
'bbb ccc'
'ddd'
'eee'
```

4. Modify this method [10 points]

```
def load(filename)
    username = name =  password = sex = nil
    friends = {}
    read_line(filename) do |line|
       args = line.scan(/[^\s]+/)
       case args[0]
          # YOUR CODE HERE
       end
   end
end
```

so that it can read a database file that looks like this:

```
##################################################
USERNAME  lonelygirl13
NAME      Daisy
SEX       Female
   PASSWORD  sailorboy
FRIENDS   nopants/dated cheeseboy/random

END
```

and returns an array like this:

```
   ["lonelygirl13","Daisy","sailorboy","Female",
     {"nopants"=>"dated","cheeseboy"=>"random"}]
```

You can assume that the input file is well-formed, so there's no need to check that the data is OK. `read_line` is as defined earlier, i.e. any line containing only whitespace should be ignored, leading whitespace should be ignored, any line starting with a `#` should be ignored. The `FRIENDS` entry can have arbitrarily many `username/how_we_met`-pairs.

4 Arrays and Hashes

1. In a Facebook system every friend relationship is defined by a short word, one of `"dated"`, `"family"`, `"friend"`, `"work"`, `"random"`, `"group"`. However, we want to present this information to the user in a more readable way. Therefore, define a hashtable [5 points]

```
FRIEND_OPTIONS = {YOUR CODE HERE}
```

such that each of the strings above are mapped to `"dated", "in my family", "met through a friend", "worked together", "met randomly", "in a group together"` respectively.

2. Write a method `square(a)` which takes a list of integers as input and returns a new list with the numbers squared. [5 points]

```
> b = square([1,2,3,4,5])
> puts b
1
4
9
16
25
```

3. Write a method `square!(a)` which takes a list of integers as input and squares the elements. The method should return `nil`. The original array should be overwritten with the new one. [5 points]

```
> a = [1,2,3,4,5]
> square!(a)
> puts a
1
4
9
16
25
```

4. Write a method `square?(a)` which returns `true` if `a` is a list of square integer values (16,64,etc.), false otherwise. [5 points]

```
> puts square?([1,4,9,16])
true
> puts square?([1,4,9,15])
false
```

5 Blocks and Iterators

1. Implement methods `map1`[1], `filter`, and `foldr`, corresponding to their Haskell namesakes, but this time in Ruby! Here is the class definition: [10 points]

```
class Array
   def Array.map1(a)
      ...
   end

   def Array.filter(a)
      ...
   end
```

[1] Since there's already a `map` function in the `Array` class, we name ours`map1`.

```
  def Array.foldr(a,z)
    ...
  end
end
```

Each method is passed an array `a` as input and returns a new array as output. In Haskell these higher-order functions would also be passed a function as argument, but here in Ruby they're instead passed a block. The `foldr` method also has an argument `z`, the starting value.

For this problem you (obviously!) cannot use any of the standard Ruby methods that implement functions corresponding to `map1`, `filter`, and `foldr`.

(a) Write the `Array.map1` method. This example

```
a = Array.map1([1,2,3]) do |x|
       x+1
    end
puts a
```

should print out

```
2
3
4
```

(b) Write the `Array.filter` method. This example

```
a = Array.filter([1,2,3,4,5]) do |x|
       x % 2 == 0
    end
puts a
```

should print out

```
2
4
```

(c) Write the `Array.foldr` method. These examples

```
a = Array.foldr([1,2,3,4,5],0) do |x,z|
       x+z
    end
puts a
a = Array.foldr([1,2,3,4,5],0) do |x,z|
       x-z
    end
puts a
a = Array.foldr(["aaa","bbb","ccc"],"") do |x,z|
       x+z
    end
puts a
```

should print out

```
15
3
aaabbbccc
```

2. Assume that we have a database describing who friends with whom in a Facebook-like system:[10 points]

```
$database = {
   "bluehat"      => ["nopants","greenhat","redhat"],
   "lonelygirl13" => ["nopants","cheeseboy"],
   "nopants"      => ["bluehat","lonelygirl13","greenhat","cheeseboy","redhat"],
   "greenhat"     => ["bluehat","nopants","redhat"],
   "redhat"       => ["bluehat","nopants","greenhat"],
   "cheeseboy"    => ["lonelygirl13","nopants"]
}
```

Here, user `bluehat` has three friends, users `"nopants"`, `"greenhat"`, and `"redhat"`.

Write a Breadth First Search routine that traverses this graph printing out each level on a separate line:

```
def BFS(queue, visited)
   YOUR CODE HERE
end

def levels(username)
   visited = {username=>true}
   queue = [username]
   BFS(queue, visited)
end
```

For example, starting from `bluehat`, the call

```
levels("bluehat")
```

we should print

```
bluehat
nopants greenhat redhat
lonelygirl13 cheeseboy
```

3. Extend the code above to instead generate the users in breadth-first order: [5 points]

```
def BFS2(queue, visited)
    ... yield ...
end

def levels2(username)
   visited = {username=>true}
   queue = [username]
   ...
end
```

Given that, this code

```
levels2("bluehat") do |user|
   puts user
end
```

should produce

```
bluehat
nopants
greenhat
redhat
lonelygirl13
cheeseboy
```

4. Finally, extend the code from above so that `levels3("bluehat")` generates pairs of (`username,level`):[5 points]

```
def BFS3(queue, visited, level)
   ...
   yield friend,level
   ...
end

def levels3(username)
   ...
end
```

This code

```
levels3("bluehat") do |user,level|
   print(user," => ",level,"\n")
end
```

should therefore produce:

```
bluehat => 0
nopants => 1
greenhat => 1
redhat => 1
lonelygirl13 => 2
cheeseboy => 2
```

6 372book — The `User` class [10 points]

In the next assignment you will be writing a minimalistic Facebook-clone (we will call it 372book) using Ruby and CGI scripting. To get started, you should write two classes `User` and `Database`.

The `User` encapsulates the data that is know about one 372book user, namely their unique username (as defined above), their real name, their sex `Male` or `Female`, their password (as defined above), and their set of friends. Friends are defined as a hashtable mapping the friend's username to the way in which the two friends met.

```
class User
   attr_reader YOUR CODE HERE
   attr_writer YOUR CODE HERE

   def initialize(username,name,friends,sex,password)
       YOUR CODE HERE
   end

   def to_out
      YOUR CODE HERE
   end
end
```

The `initialize` method creates a new `User` object. `username`, `name`, `sex`, `password` are all strings, whereas `friends` is a hashtable mapping strings to strings.

The `to_out` method returns the information in a `User` object as a string. For example, this object

```
daisy = User.new("lonelygirl13", "Daisy Duck",
                 {"nopants"=>"dated", "cheeseboy"=>"random"},
                 "Female", "sailorboy")
```

should be formated like this:

```
USERNAME  lonelygirl13
NAME      "Daisy Duck"
SEX       Female
PASSWORD  sailorboy
FRIENDS   nopants/dated cheeseboy/random
END
```

The order between the friends is not important.

7 372book — The `Database` class [10 points]

Finally, you should implement the `Database` class. In our 372book system this class encapsulates a hashtable that maps user names to a corresponding object of the `User` class. Between executions the database resides on a text file, but every time you need to operate on it (to look up a user, add a new user, or delete a user), you have to load the file from the disk into memory. Every time you've changed something (adding a user, for example), the database has to be written back to disk.

The `load()` and `save()` methods load a database from a text file and saves it back into the same file, in the same format. For example, this database file

```
USERNAME  bluehat
NAME      "Dewey Duck"
SEX       Male
PASSWORD  wood
FRIENDS   nopants/family greenhat/family redhat/family
END

USERNAME  lonelygirl13
NAME      "Daisy Duck"
SEX       Female
PASSWORD  sailorboy
FRIENDS   nopants/dated cheeseboy/random
END
```

should be loaded into a data structure that's essentially this:

```
db = {
   "lonelygirl13" => User.new("lonelygirl13", "Daisy Duck",
                      {"nopants"=>"dated", "cheeseboy"=>"random"},
                      "Female", "sailorboy")
   "bluehat"      => User.new("bluehat", "Dewey Duck",
                      {"nopants"=>"family", "greenhat"=>"family",
                       "redhat"=>"family"},  "Male", "wood")
}
```

Here is the class definition with the methods you should implement:

```
$FILE = "database.txt"
class Database

   def initialize
   def [] (name)
   def []= (name,user)

   def users()
      ... yield u ...
   end

   def exists_user?(name)

   def read_line (database)
   def split_line (line)
   def load()
   def save()
end
```

The `users()` method is an iterator that generates the user names of all the users in the database. The `[]` method returns the `User` object associated with a particular `name`. The `[]=` method associates a `User` object with a particular name.

Thus, you can build the database above like this:

```
d = Database.new
d["lonelygirl13"] = User.new("lonelygirl13", "Daisy Duck",
                   {"nopants"=>"dated", "cheeseboy"=>"random"},
                   "Female", "sailorboy")
d["bluehat"] = User.new("bluehat", "Dewey Duck",
                   {"nopants"=>"family", "greenhat"=>"family",
                    "redhat"=>"family"},  "Male", "wood")
```

And query it like this:

```
puts d["lonelygirl13"].to_out
d.users {|x| puts x}
```

Which should produce the following output:

```
USERNAME  lonelygirl13
NAME      "Daisy Duck"
SEX       Female
PASSWORD  sailorboy
FRIENDS   nopants/dated cheeseboy/random
END
bluehat
lonelygirl13
```

8 Submission and Assessment

The deadline for this assignment is noon, Wed Nov 23. It is worth 5% of your final grade.

You should submit the assignment electronically using `d2l.arizona.edu`

Don't show your code to anyone, don't read anyone else's code, don't discuss the details of your code with anyone. If you need help with the assignment see the instructor or the TA.