Homework 7

Due 18 April

Handout 18
CSCI 334: Spring, 2017

What To Turn In

Please hand in work in two pieces, one for the Problems and one for the Pair Programming:

Problems: Turn in handwritten or typed answers by the due date. Be sure your work is stapled and that your answers are clearly marked and in the correct order.

Pair Programming: This part involves writing Scala code. *You are required to work with a partner on it.* You are welcome to choose your own partner, but I can also assist in matching pairs — simply send me email and I will pair you with someone else also looking for a partner. Please do not let finding a partner go until the last minute.

There is also an optional Squeak and Smalltalk tutorial if you would like to experiment a bit with that language.

Reading

1. **(Required)** Mitchell, Chapters 10–11
2. **(Required)** The Scala Tutorial, available on the CS334 Links page.
3. **(Optional)** The Smalltalk Tutorial, available on the CS334 Handouts page.

Self Check

S1. Smalltalk Run-time Structures

Mitchell, Problem 11.4

The given conversions between Cartesian and polar coordinates work for any point (x, y), where $x \geq 0$ and $y > 0$. Do not worry about points where $x < 0$ or $y \leq 0$. The figure P.11.4.1 appears on page 332.

You should try to write reasonably accurate Smalltalk code for part (b), but you do *not* need to use Squeak, although you may give it a try if you wish. A few details if you do:

- You may wish to run through the first part of the optional Squeak tutorial first to get down the basics of writing Smalltalk code.
- Put the classes in the "My-Stuff" category you made for the tutorial, and name the classes `MyPoint` and `MyPolarPoint` to avoid conflicts with predefined classes.
- To add Class methods, click on the "class" button to the right of the "?" button in the System Browser. Click on the "instance" button to switch back to instance methods.
- There is a missing `.` at the end of the line "`x ← xCoordinate`" in the book.
- Use `sqrt` and `arcTan` methods for the math operations.

Problems

Q1. (*10 points*) ..... Removing a Method

Mitchell, Problem 11.7

Q2. (*15 points*) ..... Protocol Conformance

Mitchell, Problem 11.6

(You will find it useful to answer Problem 11.7 first before working on this one.)

Q3. (*10 points*) ..... Subtyping and Binary Methods

Mitchell, Problem 11.8

Q4. (*15 points*) ..... Delegation-Based OO Languages

Mitchell, Problem 11.9

Pair Programming

P1. (*20 points*) ..... The Happy Herd

The next few quesions ask you to implement several small programs in Scala. Start early — as I'm sure you've learned this semester, new languages can be a little tricky to grasp...

Scala in Lab. The "scala" command on the Unix machines will give you a "read-eval-print" loop, as in Lisp and ML. You can also compile and run a whole file as follows. Suppose file "A.scala" contains:

```
object A {
  def main(args : Array[String]) : Unit = {
    println(args(0));
  }
}
```

You can compile the program with "`scalac A.scala`", and then run it (and provide command-line arguments) with "`scala A moo cow`".

Resources. I have left a number of Scala books on the bookshelf in the back corner of lab. You may use them in lab, but please do not remove them the lab.

There is also plenty of very detailed information available online (e.g., `http://www.scala-lang.org` — just web search for "Scala Language"). I suggest that you look at tutorial-style descriptions of the features of interest as well as the Scala Language Specification for some of the specifics.

There is extensive online documentation for the Scala libraries at:

```
http://www.scala-lang.org/api/
```

In this first question we'll use Scala answer a few questions about cows. Specifically the herd at Cricket Creek Farm...

(a) First, write a program to read in and print out the data in the file "cows.txt" from the handouts page. Each line contains the id, name, and daily milk production of a cow from the herd. (I've also included a "cows-short.txt" file that may be useful while debugging.)

The program should be in a file called "Cows.scala" that includes a single `object` definition. Recall that `objects` are like `classes`, except that only a single instance is created.

One useful snippet of code is the following line.

```
val lines = scala.io.Source.fromFile("cows.txt").getLines();
```

We will use this to read the file. Try this out in the Scala interpreter. What type does `lines` have? For convenience in subsequent processing, it will be useful to convert `line` into a list:

```
val data = lines.toList;
```

Print out the list and verify you are successfully reading all the data. Use a `for` loop. For loops in Scala follow a familiar syntax:

```
scala> for (i <- 1 to 3) println(i);
1
2
3
```

(b) Print the `data` again, using the `foreach` method on lists.

(c) The `for` construct lets you do many other things as well, such as selectively filtering out the elements while iterating. For example:

```
scala> for (i <- 1 to 5 if i%2==0) println(i);
2
4
```

Use such a `for` list to print all cows containing "s" in their name. Make the test be case insensitive. Scala Strings support all of the same string operations as Java Strings. A few useful ones here and below:

```
def String {
  def contains(str : String) : Bool
  def startsWith(str : String ) : Bool
  def toLowerCase() : String
  def toUpperCase() : String

  // split breaks up a line into pieces separated by separator.
  //   For ex:   "A,B,C".split(",")  ->  ["A", "B", "C"]
  def split(separator : String) : Array[String]
}
```

(d) Now print all cows containing "s" but not "h". Multiple `if` clauses can be chained together, as in "`1 to 10 if i%2==0 if i%3==0`".

(e) Scala also supports lisp comprehensions:

```
val list = ...;
println (for (x <- list if ...) yield f(x));
```

Show an example of list comprehensions by computing something about the data with one. (You may need to look up list comprehensions in the documentation for more detail...)

(f) Next, define a new clss in "Cows.scala" to store one cow, its id, and its daily milk production.

```
class Cow(s : String) {
  def id = ...
  def name = ...
  def milk = ...
  override def toString() = {
    ...
  }
}
```

It takes in a string of the form "*id*,*name*,*milk*" from the data file and provides the three functions shown. For `toString`, you may find formatting code like "`"%5d ".format(n)`" handy – it formats the number n as a string and pads it to 5 characters.

Use a map operation on `data` to convert it from a list of strings to a list of `Cows`. Print the `data` and makes sure it works.

(g) Use a list comprehension to print all cows who produce more then 4 gallons of milk per day.

(h) Use the `sortWith` method on `Lists` to sort the cows by id. Also use `foldLeft` to tally up the milk production for the whole herd.

```
class List[A] {
  def sortWith (lt: (A, A) => Boolean) : List[A]
  def foldLeft [B] (z: B)(f: (B, A) => B) : B
}
```

Note that `foldLeft` is a polymorphic method with type parameter `B`. In your case, both `A` and `B` will be `Int`. Also, `foldLeft` is curried, so you must call it specially, as in:

```
val list : List[Int] = ...;
val n : Int = ...;
list.foldLeft (n) ( (x: Int, elem: Int) => ... )
```

(i) Finally, use the `maxBy` and `minBy` methods on your list of cows to find the cows with the highest and lowest daily milk production.

(j) Submit "Cows.scala" with turnin.

P2. (*15 points*) ..... Ahoy, World!

You'll now learn to speak like a pirate, with the help of Scala maps and a `Translator` class... The program will take in an English sentence and convert it into pirate. For example, typing in

"pardon, where is the pub?"

gives you

"avast, whar be th' Skull & Scuppers?"

The handouts page contains a "Pirate.scala" file to start with. You will be responsible for implementating a `Translator` class, reading in the priate dictionary, and processing the user input. It will be easiest to proceed in the following steps:

(a) First, complete the `Translator` class. It has the following signature:

```
class Translator {
  // Add the given translation from an english word to a pirate word
  def += (english : String, pirate : String) : Unit

  // Look up the given english word.  If it is in the dictionary, return the
  //    pirate equivalent.  Otherwise, just return english.
  def apply(english : String) : String
```

```
  // Print the dictionary to stdout
  override def toString() : String
}
```

Note that we're overloading the += and () operators for `Translator`. Thus, you use a Translator object as follows:

```
val pirate = new Translator();
pirate += ("hello", "ahoy");
..
val s = pirate("hello");
```

If "hello" is in the dictionary, its pirate-translation is returned. Otherwise, your translator should return the word passed in. Any non-word should also just be returned. Thus:

```
pirate("hello")   ==>  "hello"
pirate("moo")     ==>  "moo"
pirate(".")       ==>  "."
```

When writing `apply`, use the `get` method on `map` and pattern matching to handle the `Option` type it returns. (See class notes / tutorial for details on `Option`.)

Finish the definition of `Translator` using a Scala map instance variable. To write `toString`, you may find it handy to look at the `mkString` methods of the Scala Map classes.

Add a few lines to the Pirate `main` method to test your translator.

(b) Now, read in the full pirate dictionary from the "pirate.txt" data file, and print out the resulting translator.

(c) Once you have the translator built, uncomment the lines in `main` that process standard input, and process the text the user types in. There are a few sample sentences on the handouts page. Here is an example:

```
Stephen-Freund:~/scala] cat sentence1.txt
pardon, where is the pub?
I'm off to the old buried treasure.

Stephen-Freund:~/scala] scala Pirate < sentence1.txt
avast, whar be th' Skull & Scuppers?
I'm off to th' barnacle-covered buried treasure.
```

(d) Submit "Pirate.scala" with turnin.

P3. (*30 points*) ..... Argh, Expressions Matey

In Scala, algebraic datatypes can be defined with the use of abstract classes and case classes. Consider the following algebraic data type for expressions:

```
sealed abstract class Expr
case class Variable(name: Symbol) extends Expr
case class Constant(x: Double) extends Expr
case class Sum(l: Expr, r: Expr) extends Expr
case class Product(l: Expr, r: Expr) extends Expr
case class Power(b: Expr, e: Expr) extends Expr
```

This Scala code is equivalent to the following definition in ML:

```
data Expr =
     Variable of Symbol
```

```
| Constant of double
| Sum of Expr * Expr
| Product of Expr * Expr
| Power of Expr * Expr
```

Have a look at the starter code in "Expressions.scala" to see an example of Scala-style pattern matching on case classes.

(a) Write a function that takes the derivative of an expression with respect to a given variable. Your function should have the following signature:

```
def derive(e: Expr, s: Symbol): Expr
```

Your function does not have to take the derivative of Powers with non-constant exponents. It is acceptable to throw an exception in that circumstance.

Also,you'll likely need the Chain Rule for the Power case — `https://www.wyzant.com/resources/lessons/math/calculus/differentiation/chain_rule`. If your calculus is a bit rusty, you may further restrict your code to handle only cases where the base is a variable or constant.

Note that in Scala, Symbols can be declared by using a single quote before the name of the symbol, as such:

```
scala> 'x
res0: Symbol = 'x

scala> 'y
res1: Symbol = 'y

scala> 'abc
res2: Symbol = 'abc
```

(b) Write a function that evaluates a given expression in a given environment. An environment is just a mapping from symbols to values for those symbols. Your function should have the following signature:

```
def eval(e: Expr, env: Map[Symbol, Double]): Double
```

If a variable in the expression is not in the given environment, you should throw an exception.

(c) Write a function that when given an expression reduces that expression to its simplest form. Your function should have the following signature:

```
def simplify(e: Expr): Expr
```

For example,

```
simplify(Sum(Variable('x),Constant(0)))
```

should return `Variable('x)`. Your function need not be exhaustive – just implement four or five interesting cases.

(d) Submit "Expressions.scala" with turnin.

Notes: In order to make the task of writing tests easier, we provide an expression parser. The expression parser takes a string and returns its corresponding `Expr`. The expression parser can be invoked on a string `str` like so: `Expr(str)`. For example, to demonstrate that your simplifier knows about the additive identity of zero, you might write the following test:

```
assertEquals(Expr("x"), simplify(Expr("x + 0")))
```

The syntax that the expression parser accepts can be expressed by the following grammar:

```
          expr := sum
           sum := product { ("+" | "-") product }
       product := power { "*" power }
         power := factor [ "^" factor ]
        factor := "(" expr ")" | variable | constant
      variable := ident constant := floatLit
      floatLit := [ "-" ] positiveFloat
positiveFloat := numericLit [ "." [ numericLit ] ]
```

What To Turn In for the Programs.

- Your code for these questions should be documented — comments have the form "/* comment */" — and include the names of both partners at the top.
- One of each pair should include a printout of your Scala files *separate* from the answers to the written problems.
- Also, one of each pair should submit electronic copies with the command "`turnin -c 334` *file*", where *file* is the name of the file you wish to submit. Be sure to submit all three files for the Scala questions: "Dennys.scala", "Pirate.scala", and "Expressions.scala".

  You may submit files more than once if you find a mistake or wish to change what you submitted the first time. Again, only one of each pair needs to submit the code.

P4. (*15 points*) ..... Smalltalk and Squeak Tutorial (Optional)

This question is optional, but using Squeak can be a pretty interesting experience if are curious.

Go through the Smalltalk "BankAccount" tutorial from the handouts page on the Unix machines in lab. This tutorial guides you through defining and using a simple class in Squeak, and it should hopefully give you a feel for the designers' vision.

The goal is to experience Squeak enough to appreciate of it is all about (and to have an opinion about it).

A great deal more about Squeak can be found at `http://www.squeak.org`.

Here are few important details:

(a) To run Squeak, please follow these steps:

- Squeak needs to create and store a number of large files for you. To avoid filling up the student disk containing your home directory, please use a directory on `/home/scratch` for this problem. There is already a directory on that disk with the same name as your Unix id. Change to that directory, as in:

  ```
  cd /home/scratch/09abc
  ```
- In that directory, run the following command to copy thq Squeak image file to the scratch directory:

  ```
  cp -r ~freund/share/squeak-image .
  ```

  Then, run squeak:

  ```
  squeak squeak-image/squeak3.9.image
  ```

  Squeak will then use the "image" file in that directory. The image file contains the environment's code, plus any additions or modifications you have made to it. As you go through the tutorial and before you exit, you should select "save" from the "screen menu." This will save any changes you made to the image on disk.
- You can run Squeak using your modified image in the future by going back to your Squeak directory (ie, `/home/scratch/09abc`) and running the command `squeak`.

(b) Due to differences in the various Unix window managers installed in the lab, you may need to:

- Use the middle mouse button (or possibly Ctrl-Click) when the tutorial refers to the right mouse button, and
- Use the right mouse button when the tutorial asks you to `Alt-Click`.

(c) If you prefer, you can download and install Squeak on a Mac and PC, using the instructions on the Squeak website.

(d) **Do not worry about submitted the code for this part. Instead, include in your written answers and brief reflection on your experience:** What did you like? Not like? Do you think Squeak is the right model for interacting with your computer? Why or why not? A few sentences or short paragraph is sufficient.

If you want to explore Smalltalk further, I highly recommend the "Morphic" tutorial at

`http://static.squeak.org/tutorials/morphic-tutorial-1.html`

It gives a nice introduction to how to extend and add graphical objects, etc. to the Squeak environment. (The wiki `http://http://wiki.squeak.org/squeak` has many other resources you may also find interesting/entertaining.)