Homework 7

Due Wednesday, 18 April

Handout 24
CSCI 334: Spring 2018

Notes

This homework has two types of problems:

Problems: You will turn in answers to these questions.

Pair Programming: This part involves writing both ML and Java code. You are required to work with a partner for this section. I will assume that you plan to work with the same partner you worked with on the last assignment unless you email me (dbarowy@cs.williams.edu) with your partner's name **by the evening of Wednesday, April 11**.

Turn-In Instructions

You must turn in your work for this assignment in two parts, one for the Problems part and one for the Pair Programming part. You will be assigned two GitHub repositories.

Note that this assignment should be anonymized: neither your name nor your partner's name should appear in any of the files submitted with this assignment except the file "`collaborators.txt`" (see below).

Problems: Problem sets should be typed up using LaTeX and submitted using your `cs334_hw7_<username>` repository. For example, if your GitHub username is `dbarowy`, then your repository will be called `cs334_hw7_dbarowy`. Be sure that your work is committed and pushed to your repository by Wednesday, April 18 at 11:59pm.

If you discuss the problem set with your partner or with a study group, please be sure to include their names in a `collaborators.txt` file in your repository.

Pair Programming: Programming solutions should be typed up and submitted using your partner repository. For example, if your GitHub username is `dbarowy` and you are working with a partner whose username is `wjannen`, look for a repository called `cs334_hw7_dbarowy-wjannen` (usernames will be in alphabetical order). Be sure that your work is committed and pushed to your repository by Wednesday, April 18 at 11:59pm.

Reading

1. **(Required)** Read Mitchell, Chapter 8.1–8.2 and Chapter 10.
2. **(Required)** Read Mitchell, Chapter 12.4.1.

Problems

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

Smalltalk has a mechanism for "undefining" a method. Specifically, if a class `A` has method `m` then a programmer may cancel `m` in subclass `B` by writing

```
m:
   self shouldNotImplement
```

With the above declaration of `m` in subclass `B`, any invocation of `m` on a `B` object will result in a special error indicating that the method should not be used.

(a) (5 points) What effect does this feature of Smalltalk have on the relationship between inheritance and subtyping?

(b) (5 points) Suppose class `A` has methods `m` and `n`, and method `m` is canceled in subclass `B`. Method `n` is inherited and not changed, but method `n` sends the message `m` to `self`. What do you think happens if a `B` object `b` is sent a message `n`? There are two possible outcomes. See if you can identify both, and explain which one you think the designers of Smalltalk would have chosen and why.

Q2. (10 points) ..... Subtyping and Binary Methods

If there are no restrictions on how a method may be redefined in a subclass, then it is easy to redefine a method such that it appears unproblematic.

This problem is illustrated using the following `Point` class and `ColoredPoint` subclass.

class name	Point
class variables:	
instance variables:	xval yval

instance messages and methods	**xcoord** ↑ xval
	ycoord ↑ yval
	origin
	  xval ← 0
	  yval ← 0
	movex: dx movey: dy
	  xval ← xval + dx.
	  yval ← yval + dy
	equal: pt
	  ↑ (xval = pt xcoord & yval = pt ycoord)

class name	ColoredPoint
class variables:	
instance variables:	color
instance messages and methods:	**color** ↑ color **changecolor: newc** color ← newc **equal: cpt** ↑ (xval = cpt xcoord & yval = cpt ycoord & color = cpt color)

The important part to notice is the way that `equal` is redefined in the `ColoredPoint` class. This change would not be allowed in many other languages, but is allowed in Smalltalk. The intuitive reason for redefining `equal` is that two colored points are equal only if they have the same coordinates and are the same color.

Why is this redefinition a bad idea? `p1 equal:p2` will not produce a runtime error when both `p1` and `p2` are either `Point` or `ColoredPoint` objects, but it may fail when one is a `Point` and the other is a `ColoredPoint`. It helps to consider all four combinations of `p1` and `p2` as `Point`s and `ColoredPoint`s, and explain briefly how each message is interpreted.

Hint: Recall that the main principle associated with subtyping is substitutivity: if `A` is a subtype of `B`, then wherever a `B` object is required in a program, an `A` object may be used instead without producing a type error.

Q3. (8 points) ..... Function Subtyping

You will need to read Mitchell, 12.4.1 on pp. 355–356 in order to answer this question.

Assume that `A` $<:$ `B` and `B` $<:$ `C`. Which of the following subtype relationships involving the function type `B` $\rightarrow$ `B` hold in principle?

(a) (`B` $\rightarrow$ `B`) $<:$ (`B` $\rightarrow$ `B`)

(b) $(\texttt{B} \to \texttt{A}) <: (\texttt{B} \to \texttt{B})$

(c) $(\texttt{B} \to \texttt{C}) <: (\texttt{B} \to \texttt{B})$

(d) $(\texttt{C} \to \texttt{B}) <: (\texttt{B} \to \texttt{B})$

(e) $(\texttt{A} \to \texttt{B}) <: (\texttt{B} \to \texttt{B})$

(f) $(\texttt{C} \to \texttt{A}) <: (\texttt{B} \to \texttt{B})$

(g) $(\texttt{A} \to \texttt{A}) <: (\texttt{B} \to \texttt{B})$

(h) $(\texttt{C} \to \texttt{C}) <: (\texttt{B} \to \texttt{B})$

Q4. (10 points) ..... Smalltalk Implementation Decisions

In Smalltalk, each class contains a pointer to the class template. This template stores the names of all the instance variables that belong to objects created by the class.

(a) (5 points) The names of the methods are stored next to the method pointers. But instance variables are different. Why are the names of instance variables stored in the class, instead of in the objects (next to the values for the instance variables)?

(b) (5 points) Each class's method dictionary only stores the names of the methods explicitly written for that class; inherited methods are found by searching up the superclass pointers at run-time. What optimization could be done if a subclass contained all of the methods of its superclass in its method dictionary? What are some of the advantages and disadvantages of this optimization?

Pair Programming

P1. (20 points) ..... Continuation Passing Style

Continuation passing style is an alternative way to structure a computation that is especially useful when designing recursive algorithms. A continuation is a function that represents the evaluation of the "rest of the program" from a given point in a program.

Suppose you have the following SML code for a bubble sort algorithm:

```
fun bubble_inner [] = []
  | bubble_inner [a] = [a]
  | bubble_inner (a::b::xs) =
    if b < a then b::(bubble_inner(a::xs)) else a::(bubble_inner(b::xs));

fun bubble_outer [] = []
  | bubble_outer (x::xs) = bubble_inner (x::(bubble_outer xs));

fun bubblesort xs = bubble_outer xs;
```

The algorithm is divided into three pieces: a wrapper function called `bubblesort` that presents a simple interface for users, a function representing the "outer loop" of the sort called `bubble_outer` and a function representing the "inner loop" of the sort called `bubble_inner`. Since this is a functional program, we use liberally use recursion instead of looping constructs.

(a) Rewrite the algorithm in continuation passing style (CPS). Remember that a CPS transform usually involves

i. identifying work done before a recursive call,

ii. identifying work done after a recursive call,

iii. adding a continuation parameter `k` to the function,

iv. constructing a continuation representing the work done after the recursive call,

v. then re-writing the recursive call in tail form, passing the continuation as a parameter.

You may change the function signatures of `bubble_outer` and `bubble_inner` as required, but note that the signature for `bubblesort` must remain the same. In other words, there should be no discernable difference between the behavior of the original `bubblesort` and your CPS-transformed `bubblesort` from the user's perspective. For example, your `bubblesort` should behave exactly as follows:

```
- val xs = [3,7,1,0,0,45,1001,2,~100];
val xs = [3,7,1,0,0,45,1001,2,~100] : int list
- bubblesort xs;
val it = [~100,0,0,1,2,3,7,45,1001] : int list
```

You may not use `callcc` or `throw` to answer this part of the question. Also note that both `bubble_inner` and `bubble_outer` must be in continuation passing form.

(b) Write a second continuation passing version of `bubblesort` that uses `callcc` and `throw` instead of manually generating lambda expressions. You may find it helpful to use the following definitions:

```
val callcc = SMLofNJ.Cont.callcc;
val throw = SMLofNJ.Cont.throw;
```

Again, ensure that `bubblesort` works exactly the same way (it does not expose continuations to the user) and that both `bubble_outer` and `bubble_inner` are in CPS.

(c) Name one reason why we might want to represent a program in continuation passing style.

(d) What is the advantage of the `callcc` version of `bubblesort` over the ordinary CPS version?

P2. (20 points) ..... The Happy Herd

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. There are 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 lots 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.

Scala libraries are extensively documented 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` found in your repository. 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 `object`s are like `class`es, 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 list 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 calss 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 `Cow`s. 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 `List`s 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.

P3. (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?"

Your repository constains the starter code `Pirate.scala`. You will be responsible for implementing a `Translator` class, reading in the pirate 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(".")       ==>  "."
```

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

(c) Once you have the translator built, create a `PirateInterpreter.scala` program by changing the lines in `main` that process standard input, instead interactively processing text the user types in. There are a few sample sentences in your repository. Here is an example:

```
$ cat sentence1.txt
pardon, where is the pub?
I'm off to the old buried treasure.

$ scala Pirate < sentence1.txt
avast, whar be th' Skull & Scuppers?
I'm off to th' barnacle-covered buried treasure.
```