Programming Principles

Final Exam

Friday, December 19th 2014

First Name: _____

Last Name: _____

Your points are *precious*, don't let them go to waste!

Your Name Work that can't be attributed to you is lost: write your name on each sheet of the exam.

Your Time All points are not equal. Note that we do not think that all exercises have the same difficulty, even if they have the same number of points.

Your Attention The exam problems are precisely and carefully formulated: some details can be subtle. Pay attention, because if you do not understand a problem, you can not obtain full points.

Your Answers Answer each question on a separate sheet of paper.

Some Help This exam is *closed book*. We provide an appendix containing some useful functions. Take a close look at these, they may provide elegant solutions to complex problems.

Exercise	**Points**	**Points Achieved**
1	11	
2	10	
3	9	
Total	30	

Exercise 1: Comprehensions (11 points)

For-Comprehensions and Higher-Order Functions

In this exercise, we'll focus on using for-comprehensions and higher-order functions to analyze and query a collection of movies.

Consider a case class `Movie`

```
case class Movie(title: String, director: String, releaseDate: Int, leadActors: List[String])
```

where

- `title` represents the title of the movie
- `director` represents the full name of the movie's director
- `releaseDate` is an `Int` representing the year the movie was released
- `leadActors` represents a list of the full names of the lead actors for the given title.

Here is an example of a list of movies:

```
val movies: List[Movie] = List(
  Movie("The Master", "Paul Thomas Anderson", 2012,
    List("Joaquin Phoenix", "Philip Seymour Hoffman", "Amy Adams")
  ),
  Movie("Unforgiven", "Clint Eastwood", 1992,
    List("Clint Eastwood", "Gene Hackman", "Morgan Freeman")
  ),
  Movie("Magnolia", "Paul Thomas Anderson", 1999,
    List("Tom Cruise", "Philip Seymour Hoffman", "William H. Macy")
  ),
  Movie("The Big Lebowski", "Joel and Ethan Coen", 1998,
    List("Jeff Bridges", "John Goodman", "Philip Seymour Hoffman")
  ),
  Movie("Million Dollar Baby", "Clint Eastwood", 2004,
    List("Clint Eastwood", "Hilary Swank", "Morgan Freeman")
  ),
  Movie("The Grand Budapest Hotel", "Wes Anderson", 2014,
    List("Ralph Fiennes", "F. Murray Abraham", "Mathieu Amalric")
  ),
  Movie("The Good, the Bad and the Ugly", "Sergio Leone", 1966,
    List("Clint Eastwood", "Eli Wallach", "Lee Van Cleef")
  )
)
```

Part 1a

For-comprehension: (2 points) Write a for-comprehension that returns a pretty-printed list of a movie's title along with the number of years since the movie has been released, as a `List[String]`. You can use `2014` as the value for the current year. The format for this comprehension is given below:

```
TitleOfMovie, directed by NameOfDirector (YearsSinceRelease years ago)
The Master, directed by Paul Thomas Anderson (2 years ago)
```

Higher-order function (1 point) Desugar the for-comprehension from 1a into calls to higher-order functions such as `map`, `flatMap`, and `withFilter` instead.

Part 1b

For-comprehension: (2 points) Write a for-comprehension that returns a list of strings containing the title of all movies that actor Philip Seymour Hoffman starred in.

Higher-order function (1 point) Desugar the above for-comprehension from 1b into calls to higher-order functions such as `map`, `flatMap`, and `withFilter` instead.

Part 1c

For-comprehension: (2 points) Sometimes actors move on to become directors. Write a for-comprehension that goes through a list of movies, and returns a `List[(String, String)]`. Each element of this list is a pair where the first element is a director's name, and the second element is the movie title he/she starred in as an actor. (It's OK to return duplicate director/movie title pairs in your result.)

For example, a list with these elements is a valid result:

```
List(
  ("Clint Eastwood", "Unforgiven"),
  ("Clint Eastwood", "Million Dollar Baby"),
  ("Clint Eastwood", "The Good, the Bad and the Ugly"),
  ("Clint Eastwood", "Unforgiven"),
  ("Clint Eastwood", "Million Dollar Baby"),
  ("Clint Eastwood", "The Good, the Bad and the Ugly")
)
```

Part 2: Higher-Order Functions (2 points)

Returning a list of director/movie title pairs with duplicates is a bit awkward – it is more convenient to return a `Map` instead. The type of this map is `Map[String, List[String]]` where the key is the director's name and the value is a list of movie titles which that director starred in as an actor.

Use higher-order functions to operate on `movies` and return a `Map[String, List[String]]` which represents this mapping. This time, you will have to ensure that list does *not contain duplicate elements.*

Given `movies` as defined above, a possible return value is:

```
Map("Clint Eastwood" ->
  List("Unforgiven", "Million Dollar Baby", "The Good, the Bad and the Ugly")
)
```

Note: you can use intermediate values to simplify your implementation. Using meaningful variable names will also help you better understand what you are doing.

Exercise 2: Negation Normal Form (10 points)

In first-order logic, a formula is in *negation normal form* iff

- the $\neg$ (NOT) operator is only applied to variables
- and the only other operators are $\wedge$ (AND) and $\vee$ (OR).

For example, the following formula is *not* in negation normal form:

$$\neg(\neg(\neg\neg A \wedge (B \Rightarrow C) \wedge \neg C) \wedge \neg D)$$

because

- It contains an $\Rightarrow$
- The operator $\neg$ is applied to an $\wedge$, and also to another $\neg$

The following formula is logically equivalent to the previous one, and is in negation normal form:

$$(A \wedge (\neg B \vee C) \wedge \neg C) \vee D$$

Consider the case class hiearchy described in the Appendix to represent logic formulae. Write the two following functions:

```
def elimImplies(f: Formula): Formula = ???
def negationNormalForm(f: Formula): Formula = ???
```

`elimImplies` returns a new formula that is logically equivalent to the input, and does not contain any `Implies` node. To do so, the following logical equivalence can be used:

$$P \Rightarrow Q \equiv \neg P \vee Q$$

`negationNormalForm` returns a new formula that is logically equivalent to the input and is in negation normal form. The input is assumed not to contain any `Implies` node (so in real life you would chain a call to `elimImplies` and `negationNormalForm`, but you do not have to do this here). To do so, the three following logical equivalences can be used:

$$\neg(P \wedge Q) \equiv \neg P \vee \neg Q$$
$$\neg(P \vee Q) \equiv \neg P \wedge \neg Q$$
$$\neg\neg P \equiv P$$

Note: You *must not* use any mutable state in your solution. It should be functional. Use the constructs of the Scala language wisely to implement the two functions.

Exercise 3: Constraints (9 points)

Consider a simplified version of the N-queens problem, which we call the N-rooks problem. We have an $n \times n$ chess board and the goal is to place `n` rooks (tour in French) such that no two rooks threaten each other. This essentially boils down to having *exactly* one rook per row and column.

Part 1: Verify a positioning

We first seek to verify that a positioning of the rooks is correct. This function should work for all chess board dimensions. The positioning is represented as an `Array[Array[Boolean]]`. A cell in this matrix is set to `true` if a rook is at this position, `false` otherwise.

Verifying constraints for a single row/column (2 points)

First, write a function that verifies correctness for a single row:

```
def checkRow(row: Array[Boolean]): Boolean = ???
```

The function `checkRow` returns `true` if exactly one rook is in this row.

Verifying constraints for the board (2 points)

Using the above function, write the `checkPositioning` function that verifies the correctness for the whole board.

```
def checkPositioning(board: Array[Array[Boolean]]): Boolean = ???
```

The `checkPositioning` function should return `true` if the positioning of the rooks is valid:

- Exactly one rook in each row
- Exactly one rook in each column

Note: You can use for-comprehensions or higher-order functions as you please. You must *not* make use of mutable state.

Part 2: Describing the problem with constraints

We now want to describe the problem as a logical formula. Suppose we introduce one propositional variable for each square of the chess board. Then, a valuation (mapping from propositional variable to boolean value) for those variables is essentially a positioning of the rooks. If a variable is evaluated to `true`, then a rook is positioned at the corresponding square. We want to to build a formula that constrains the value of those variables such that a valuation only defines a valid positioning.

A formula for a single 1×3 row (2 points)

Suppose we restrict ourselves to a 3×3 board, and you are are given variables `p1, p2, p3`, of type `Var` as in the appendix. These are propositional variables for the first row. Using the class hierarchy in the appendix, write down a `Formula` which expresses the rooks constraint for the row. **Note**: Do not write a function that creates this `Formula`, just the complete formula itself.

```
val handWrittenFormula: Formula = ???
```

A formula for a row of arbitrary length (3 points)

Using the case class hierarchy in the appendix, implement a function that generalizes the above, but for a row of arbitrary size. You can assume the row is of length at least 3. Given an array of `Var`s as input, implement a `formulaForRow` function that constructs a formula for this row, with the requirement that exactly one element of the row can be `true`:

```
def formulaForRow(row: Array[Var]): Formula = ???
```

Appendix

An ADT to represent logic formulae

The following abstract data type (ADT) represents first-order logic formulae. It is used by some questions in this exam.

```
sealed abstract class Formula
final case class Var(name: String) extends Formula
final case class Not(p: Formula) extends Formula
final case class And(p: Formula, q: Formula) extends Formula
final case class Or(p: Formula, q: Formula) extends Formula
final case class Implies(p: Formula, q: Formula) extends Formula
```

The meaning should be straightforward. For `Implies`, the direction is `p => q`.

A non-exhaustive (but useful) API

Here are some methods from the Scala standard library that you may find useful:

on `List` (containing elements of type `T`):

- `def ++(that: List[T]): List[T]`: returns a new list containing the elements from the left hand operand followed by the elements from the right hand operand.
- `def count(p: T => Boolean): Int`: returns the number of elements in the list which satisfy predicate `p`. Returns `false` if the list is empty.
- `def distinct: List[T]`: returns a list which contains all elements of the original elements, but no duplicates.
- `def exists(p: T => Boolean): Boolean`: returns `true` if there is at least one element in the list that satisfies predicate `p`. Returns `false` if the list is empty.
- `def filter(p: T => Boolean): List[T]`: Selects all elements of this list which satisfy predicate `p`.
- `def flatMap[U](f: T => List[U]): List[U]`: builds a new list by applying function `f` to all elements of this list and concatenating the elements of the resulting lists.
- `def foldLeft[U](z: U)(op: (U, T) => U): U`: applies the binary operator `op` to a start value and all elements of the list, going left to right
- `def forall(p: T => Boolean): Boolean`: returns `true` if all elements in the list satisfy predicate `p`. Returns `true` if the list is empty.
- `def groupBy[K](f: T => K): Map[K, List[T]]`: partitions a list into a map of lists according to some discriminator function.
- `def map[U](f: T => U): List[U]`: builds a new list by applying a function to all elements of this list.
- `def reduce[T](f: (T, T) => T): T`: reduces the elements of this list using the specified associative binary operator.
- `def withFilter(p: T => Boolean): List[T]`: Selects all elements of this list which satisfy predicate `p`.
- `def zipWithIndex: List[(T, Int)]`: zips the list with its indices.

on `Array` (containing elements of type `T`):

- the functions defined on `List` above apply on `Array` as well. Substitute `List` by `Array` to recover the correct type signature.
- `def transpose[U]: Array[Array[U]]`: transposes this array if it is two-dimensional.

on `Map[K, V]` (containing keys of type `K` and values of type `V`):

- `def map[K2, V2](f: (K, V) => (K2, V2)): Map[K2, V2]`: returns a new map by applying `f` to all key-value pairs of this map.
- `def mapValues[V2](f: V => V2): Map[K, V1]`: returns a new map by applying `f` to all values of this map.

miscellaneous

- `(n until m).toList` creates a list of integers, from `n` until, and excluding `m`. It returns an empty list if `m <= n`.
- `(n to m).toList` creates a list of integers, from `n` up to, and including `m`. It returns an empty list if `m < n`.