Programming Principles

Final Exam

Friday, December 21st 2012

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.

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

Exercise 1: for-Comprehensions and De-Sugaring (10 points)

All is not well in Middle Earth. A company of exiled dwarves of the Lonely Mountain have decided to recapture their home, which has fallen prey to Smaug, a terrifying dragon. Before they are ready to wage war, however, they deem it important to find a leader who can unite them and raise an army. Being proud people, each dwarf of the company considers himself a valuable and deserving leader.

To avoid unnecessary quarrels and bloodshed, the dwarves agree that they will elect a leader after a sword forging contest. Renowned for their craftsmanship, the dwarves face each other in a one-to-one race; the quickest one to forge a sword will win (There is no need to doubt the quality of the sword, these are dwarves after all). To remove any element of chance, each dwarf competes against every other dwarf twice, a "home" and "away" duel.

During their visit to the Shire, they come to you, a Scala expert, for help. They would like you to determine all the duels that need to take place for a full tournament. For convenience, you define a type alias for a duel:

```
type Duel = (String, String)
```

Duels and Higher-Order Functions (2 points)

Write a `duels` function, which, given a list of names, produces a list of pairings/duels that should take place. Write your solutions using higher-order functions on `List`.

```
def duels(dwarves: List[String]) : List[Duel] = ???
```

Here is an example of an output :

```
scala> duels(List("Thorin", "Gloin", "Balin"))
res0: List[(String, String)] = List((Thorin,Gloin), (Thorin,Balin), (Gloin,Thorin),
                                     (Gloin,Balin), (Balin,Thorin), (Balin,Gloin))
```

Comprehending the Function (2 points)

One of the dwarves, Gloin, is a little bit dense, but also eager to learn, and he does not know of higher-order functions. He requests you for an explanation, and you kindly oblige by writing it with **for comprehensions only** :

```
def duels(dwarves: List[String]) : List[Duel] = ???
```

Recursion Galore (3 points)

Being on a roll, you decide to show off your Scala and functional programming skills. Implement the same function once again, but this time using recursion, i.e. your implementation must contain calls to `duels` on (potentially) smaller sub-lists. Hint: you can safely assume that the list contains at least 2 elements.

```
def duels(dwarves: List[String]) : List[Duel] = ???
```

Picking the Winner (3 points)

After a long and tiring tournament, the scores are out. Your Scala skills are called one final time to pick the winner. The scores are given as a list of pairs, where the first element of the pair is itself a pair describing a duel, and the second element is an integer of value `1` or `-1`. If the value is `1`, it means the first person in the pair won the duel, while a `-1` indicates a victory for the second person. For example, the following list:

```
List((("Thorin","Gloin"),1), (("Thorin","Balin"), -1), (("Gloin","Thorin"), -1),
     (("Gloin","Balin"),1), (("Balin","Thorin"), -1), (("Balin","Gloin"), 1))
```

tells us that Thorin won 3 duels, while Gloin and Balin won 2 duels each. Write a function that, given such a list, returns the name of the winner of the tournament. You can safely assume that there is only one winner, and that the list is non-empty.

```
def winner(duels: List[(Duel, Int)]): String = ???
```

Hint: you are welcome to use higher-order functions on `Map` and `List` that you have seen, for example, during your midterm.

Epilogue: As you have seen above, Thorin won the sword forging competition and led his men. The rest, as you know, is history.

Exercise 2: Lisp (10 points)

Recall the Lisp interpreter seen in class. In this exercise, your task is to write Lisp code that would run in your interpreter. Here is a summary of special forms and functions that you may need for this exercise.

- `(if c a b)`: special form which evaluates `c`, and then `a` if `c` is not `0` and `b` if `c` is `0`.
- `(cond (c1 r1) ... (cn rn) (else relse))`: special form which evaluates `c1`, then `r1` if `c1` is true, or else continues with the other clauses.
- `(cons first rest)`: constructs a list equivalent to Scala's `first::rest`. In our interpreter, `rest` *must* be a list.
- `(car lst)`: returns the first element of a given list.
- `(cdr lst)`: returns the rest of a given list (a list of all but the first element).
- `(quote x)`: returns x as a quoted expression, i.e. `(quote foo)` returns the quoted symbol `foo`, and `(quote (a b c))` returns the list equivalent to `(cons (quote a) (cons (quote b) (cons (quote c) nil)))`
- `(= a b)`: returns whether `a` and `b` are equal. In our interpreter, `a` and `b` may be numbers, symbols or even lists.
- `(lambda (p1 ... pn) body)`: creates an anonymous function.
- `def f x`: creates a definition.
- `def (f p1 ... pn) body`: syntactic sugar for defining a named function.

Implementing `filter` in Lisp (5 points)

In Lisp, implement the familiar higher-order function `filter`, which takes a list and a predicate (a function which takes an element as argument and returns a boolean), and returns a new list which contains the elements from the original list which satisfy the predicate (i.e. for which the predicate is true).

For example:

```
(filter (lambda (x) (= x 2)) (cons 1 (cons 2 (cons 3 nil))))))
```

returns the list `'(2)`.

Desugaring `and` in Lisp (5 points)

The `and` operator can be thought of as a "short-circuit" operator. That is, given two arguments `a` and `b` passed to `(and a b)`, if `and`'s first argument `a` evaluates to `0` (or false), then the entire expression will evaluate to `0` (or false) without attempting to evaluate `b`. If `a` does not evaluate to `0`, then `b` is evaluated and returned.

Our goal in this exercise is to *desugar* an `and` expression using only Lisp. That is, we'd like to translate the `and` expression into its equivalent `if` expression in Lisp.

Assume that we have a `lst` of the form `(and a b)`, where `a` and `b` do not contain a usage of `and`. Complete the following function so that `lst` gets rewritten to its expansion using `if`.

```
def (desugar-and lst) ???
```

For example, `evaluate(evaluate("(desugar-and (quote (and 1 2)))"))` should return `2`.

1. What is the expansion using `if` of `(and 1 2)`. In other words, what does `evaluate("(desguar-and (quote (and 1 2)))")` return? (1 point)

2. Implement `desugar-and`. (4 points)

Exercise 3: Memoization (10 points)

In this exercise you will implement a recursive function which evaluates Fibonacci numbers. You will then improve its performance by caching its results with a technique called *memoization*.

The `fib` method (2 points)

Write a recursive method `fib` which, given an integer `n` computes and returns the `n`-th Fibonacci number.

```
def fib(n: Int) = ???
```

The first two Fibonacci numbers are `1`. Every other Fibonacci number is the sum of the previous two Fibonacci numbers.

Use recursion in your implementation.

The `memo` operation (4 points)

Our Fibonacci method works, but it has a problem – it is very slow for larger numbers! If we take a look at the recursive calls, we can notice that some Fibonacci numbers are computed more than once:

```
fib(n) = fib(n - 1) + fib(n - 2)
       = fib(n - 2) + fib(n - 3) + fib(n - 2)
       = fib(n - 3) + fib(n - 4) + fib(n - 3) + fib(n - 3) + fib(n - 4)
       = ...
```

In fact, the number of recursive calls is exponential in the argument `n`, and this is due to the fact that we call the method `fib` multiple times with the same arguments.

If we could somehow remember that we're already evaluated a certain tuple, we could use the solution we've computed earlier and avoid computing it twice. This is called memoization.

The task of this part of the exercise is to implement the method `memo` which takes a function `f` and returns its memoized version `mf`. The memoized function `mf` internally keeps a mutable `Map` that maps function arguments to return values. Each time a memoized function `mf` is applied to some argument it uses the `Map` to check if it was previously applied to that argument. If it was, it returns the return value associated with that argument. Otherwise, it uses the original function `f` to compute the return value and returns it.

For example, the following snippet:

```
val mf = memo(fib)
mf(10)
mf(10)
```

will not recompute all the Fibonacci numbers up to `10` the second time `mf` is called. Similarly, the following snippet:

```
val memolength = memo { xs: List[Int] =>
  println("Computing...")
  xs.length
}
memolength(List(1, 2, 3, 4, 5, 6, 7))
memolength(List(1, 2, 3, 4, 5, 6, 7))
```

will print `Computing...` only once. The reason why this works is because `memolength` is a `val` – we invoke the same function value both times. Had `memolength` been a `def` this memoization wouldn't work.

Implement the method `memo`:

```
def memo[A, B](f: A => B): A => B = ???
```

You should use the `mutable.Map` implementation which supports the following operations:

- `mutable.Map[A, B]` - creates a new mutable `Map` mapping keys of type `A` to values of type `B`
- `get(key: A): Option[B]` - returns `Some(v)` if the `key` is present in the `Map`, or `None` otherwise
- `put(key: A, value: B): Option[B]` - adds a binding from `key` to `value` into the `Map` – returns `Some(v)` if `key` was previously associated with some value `v`, or `None` otherwise

Improving the complexity using the `memo` operation (4 points)

The `memo` function is useful because it helps us avoid recomputing elements of the list that we have already been computed earlier.

However, note that calling `memo(fib)` does not improve the running time of computing Fibonacci numbers if we use `memo(fib)` only once. The reason is that we've applied `memo` only at the top-level – every recursive call still calls `fib` and not the memoized version.

Use `memo` to define a recursive method `memfib` that computes a Fibonacci number, such that it does not recursively compute those Fibonacci numbers that had already been computed once. Do not use `fib` from the first part of the exercise – instead, write a new method from scratch.

```
def memfib(n: Int) = ???
```