Programming Principles

Midterm Exam

Wednesday, November 6th 2013

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.

Some help The last page of this exam contains an appendix which is useful for formulating your solutions.

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

Exercise 1: Data Abstraction - Operations on Polynomials (10 points)

In this exercise, we will focus on finding elegant ways to represent operations on polynomials. A way to model polynomials would be using

```
case class Poly(ls: List[Int])
```

Likewise, examples of polynomial expressions would be

```
val poly1 = Poly(List(3,2,1)) // 3 + 2*x + x^2

val poly2 = Poly(List(1,4))   // 1 + 4*x
```

Part 1: Operations over polynomials

We want to define a number of basic operations over polynomials. Start by defining methods for addition of polynomials, and multiplication of a polynomial with a scalar. Then, express subtraction of polynomials using the addition method.

```
def +(that: Poly): Poly = ???

def *(n: Double): Poly = ???

def -(that: Poly): Poly = ???
```

For example:

```
val poly1 = Poly(List(3,2,1)) // 3 + 2*x + x^2

val poly2 = Poly(List(1,4))   // 1 + 4*x

val result1 = poly1 + poly2     // 4 + 6*x + x^2

val result2 = poly2 * 2         // 2 + 8*x
```

Part 2: Compact polynomial representation

The representation we have been using so far is not ideal for all polynomials. Representing a polynomial such as $1 + 2 * x^{30}$ would include a lot of redundancy; the representation would be `Poly(List(1,0,0,...,2))`. An alternative would be storing only the non-zero coefficients of the polynomial, along with the position they occur at.

Given the case class

```
case class SparsePoly(repr: List[(Int,Int)])
```

implement the function

```
def toSparse(p : Poly): SparsePoly = ???
```

to create a sparse representation of a given polynomial.

For example:

```
val x = Poly(List(3, 0, 0, 0, -5)) //3 - 5 * x^4
toSparse(x) //SparsePoly((3,0),(-5,4))
```

Note: You may use higher-order functions in the solution you provide.

Part 3: Expand polynomial representation

Implement the function

```
toDense(s: SparsePoly): Poly = ???
```

which reverts a sparse polynomial to the original dense representation. For example:

```
val y = SparsePoly((3,0),(-5,4)) //3 - 5 * x^4
toDense(x) //Poly(List(3, 0, 0, 0, -5))
```

Note: For this part, assume that the list argument of a sparse polynomial is ordered by the index of the coefficient, just as in the provided example.

Exercise 2: Equational Proofs on Lists (10 points)

We define the `foldRight` and `drop` operations on `List` as:

```
def foldRight[T, Z](xs: List[T], z: Z, f: (T, Z) => Z): Z = xs match {
  case Nil => z
  case x :: xs => f(x, foldRight(xs, z, f))
}

def drop[T](xs: List[T], n: Int): List[T] = xs match {
  case Nil => Nil
  case x :: xs => if(n <= 0) x :: xs else drop(xs, n - 1)
}
```

Part 1: Length of a List as a `foldRight`

Implement the length operation on `List` by using the `foldRight` definition given above.

```
def length[T](xs: List[T]): Int = ???
```

Part 2: Proof for length

Given your definition of `length`, prove, by induction, that:

```
length(Nil)      = 0
length(x :: xs) = length(xs) + 1
```

Part 3: Proof for drop

Prove that:

```
drop(xs, length(xs)) = Nil
```

Note: Be very precise in your proofs. Clearly state which rules/axioms you use, and when/if you use the induction hypothesis.

Exercise 3: Propositional Logic (10 points)

Propositional logic is a logic on boolean formulae. We can represent quantifier-free propositional logic in Scala as follows:

```
sealed abstract class Prop
case class And(p: Prop, q: Prop) extends Prop
case class Var(id: String) extends Prop
case class Not(p: Prop) extends Prop
case object False extends Prop
def True: Prop = Not(False)
def Or(p: Prop, q: Prop): Prop = Not(And(Not(p), Not(q)))
def Iff(p: Prop, q: Prop): Prop = Or(And(p,q),And(Not(p),Not(q)))
def Implies(p: Prop, q: Prop): Prop = Or(Not(p), q)
```

We use definitions for derived constructs, such as `Or`, which is defined in terms of `Not` and `And`. This way, we keep the number of cases to handle at a minimum.

The `Var` case class is used to encode primitive propositions, such as `Var("snowing")`. We can encode a proposition such as if it's snowing, then it is cold, as `Implies(Var("snowing"), Var("cold"))`.

Part 1: Evaluation of Propositional Logic Formulae

Define a method `eval` inside of the `Prop` class, with the following signature:

```
sealed abstract class Prop {
  def eval(env: Map[Var,Boolean]): Boolean = ???
}
```

You can assume that the parameter `env` contains an entry for each primitive proposition in the `this` formula.

For example,

```
Implies(Var("snowing"), Var("cold")).eval(
  Map(Var("snowing") -> false,
      Var("cold") -> true))
```

should return `true`, while

```
Implies(Var("snowing"), Var("cold")).eval(
  Map(Var("snowing") -> true,
      Var("cold") -> false))
```

should return `false`.

Part 2: Free variables

The *support* of a logic formula is the set of its free variables or primitive propositions – that is, those that need to be in the initial environment when evaluating the formula. Define the `support` method in the top-level `Prop` class.

```
sealed abstract class Prop {
  // ...
  def support: List[Var] = ???
}
```

Here is an example:

```
> Var("x").support
> List(Var("x"))

> And(Var("x"), Var("y")).support
> List(Var("x"), Var("y"))
```

Part 3: Truth Tables

An environment is a particular assignment of boolean values to the support of a proposition. We type the environment as `Map[Var, Boolean]`. For example, one environment for `And(Var("x"), Var("y"))` is

```
Map(Var("x") -> true, Var("y") -> false)
```

A truth table of a formula is a list of all possible environments and the result of their evaluation. In Scala, we represent a truth table with the type `List[(Map[Var,Boolean], Boolean)]`. Define the `truthTable` method in the top-level `Prop` class. Use the `inner` function to recursively generate all the possible environments for a list of variables.

```
sealed abstract class Prop {
  // ...
  def truthTable: List[(Map[Var,Boolean],Boolean)] = {

    def inner(ls: List[Var]): List[Map[Var, Boolean]] = ???
  }
}
```

Here are some examples of truth tables.

Truth table for `Var("x")`:

```
x=F; F
x=T; T
```

Truth table for `And(Var("x"), Var("y"))`:

```
y=F;x=F; F
y=T;x=F; F
y=F;x=T; F
y=T;x=T; T
```

Truth table for `Or(Var("x"), Not(Var("x")))`:

```
x=F; T
x=T; T
```

Part 4: Satisfiable and Tautology

Now, using the truth table, you can easily define whether a proposition is *satisfiable* (true for at least one row in the table) or a *tautology* (true for each row in the table). Define these methods in the `Prop` class:

```
sealed abstract class Prop {
  // ...
  def satisfiable: Boolean = ???
  def tautology: Boolean = ???
}
```

Appendix: Scala Standard Library Methods

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

- on List:
  - `xs.contains(x)`: tests whether `xs` contains the element `x`.
  - `xs.exists(p)`: whether any elements of the list `xs` satisfy the predicate `p`
  - `xs.filter(p)`: returns all elements from `xs` that satisfy the predicate `p`
  - `xs.forall(p)`: whether all elements of the list `xs` satisfy the predicate `p`
  - `xs.map(f)`: applies f to all elements of the list `xs`
  - `xs.zip(ys)`: returns a list formed from this `xs` and `ys` by combining corresponding elements in pairs
  - `xs.zipWithIndex`: zips `xs` with its position indices
  - `xs.zipAll(ys, left, right)`: returns a list formed from `xs` and `ys` by combining corresponding elements in pairs. `left` and `right` are default elements which fill holes.
- on Map: `m.updated(k,v)`: returns a new map which is like `m` but with `k` mapping to `v`