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.

<table>
<thead>
<tr>
<th>Exercise</th>
<th>Points</th>
<th>Points Achieved</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>30</td>
<td></td>
</tr>
</tbody>
</table>
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

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

Likewise, examples of polynomial expressions would be

```scala
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.

```scala
def +(that: Poly): Poly = ???
def *(n: Double): Poly = ???
def -(that: Poly): Poly = ???
```

For example:

```scala
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 $1 + 2 \cdot x^{30}$ 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

```scala
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:

\[
\begin{align*}
def\ \text{foldRight}[T,\ Z](xs: List[T],\ z: Z,\ f: (T, Z) \Rightarrow Z): Z &= \text{xs match} \{ \\
&\text{case Nil} \Rightarrow z \\
&\text{case x :: xs} \Rightarrow f(x, \text{foldRight(xs, z, f)}) \\
&\} \\
def\ \text{drop}[T](xs: List[T],\ n: Int): List[T] &= \text{xs match} \{ \\
&\text{case Nil} \Rightarrow Nil \\
&\text{case x :: xs} \Rightarrow \text{if(n <= 0) x :: xs else drop(xs, n - 1)} \\
&\}
\end{align*}
\]

Part 1: Length of a List as a foldRight

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

\[
\begin{align*}
def\ \text{length}[T](xs: List[T]): Int &= ???
\end{align*}
\]

Part 2: Proof for length

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

\[
\begin{align*}
\text{length( Nil)} &= 0 \\
\text{length(x :: xs)} &= \text{length(xs)} + 1
\end{align*}
\]

Part 3: Proof for drop

Prove that:

\[
\begin{align*}
\text{drop(xs, length(xs))} &= \text{Nil}
\end{align*}
\]

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:

```scala
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:

```scala
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,

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

should return true, while

```scala
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.

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

Here is an example:

```scala
> 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.

```scala
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"))`:
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:

```scala
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`