Lab 5. Variance and implicits

5.1. Variance

We will use a couple of animal types represented as classes for the following exercises.

```
class Animal(name: String) {
    override def toString: String = name
}
case class Cat(name: String) extends Animal(name)
case class Dog(name: String) extends Animal(name)
```

Covariance

5.1.1. We introduce a carrier class. Modify the type signature so that it allows for animal subtypes.

```
class Carrier[T <: Animal](val content: T)
```

5.1.2. Evaluate the following and give an explanation as to why they work or not.

```
val catCarrier: Carrier[Cat] = new Carrier[Cat](Cat("Felix"))
val animalCarrier: Carrier[Animal] = catCarrier
val dogCarrier: Carrier[Dog] = new Carrier[Dog](Cat("Merv"))
```

Contravariance

We introduce a veterinarian class. Consider why this class should be contravariant in `T`.

```
class Vet[T <: Animal] {
    def treat(patient: T): String = {
        "Can treat " + patient.toString
    }
}
```

5.1.3. Evaluate the following and give an explanation as to why they work or not.

```
val generalVet: Vet[Animal] = new Vet[Animal]
```

```
val dogVet: Vet[Dog] = generalVet
dogVet.treat(Cat("Bob"))
val catVet: Vet[Cat] = new Vet[Dog]
```

We'd like to be able to compare animals through the help of a `Comparator` class.

5.1.4. Implement a comparator class that compares animals by their name. Determine the type parameter and whether it should be covariant/contravariant/invariant.

```
enum Ord:
    case LT, EQ, GT

class Comparator[???] {
    def compare(o1: T, o2: T): Ord = ???
}
```

5.1.5. Determine which animal would come first alphabetically from a list of animals, using the implemented comparator.

```
def detFirst(lst: List[Animal]): Animal = {
    val comparator = new Comparator[Animal]
    ???
}
```

5.1.6. analyze the following function signature and come up with a function that could be passed, with sub/supertypes that differ from the signature ones. (Hint: What trait does List extend?)

```
def apply(g: List[Animal] => Animal)(lst: List[Animal]): Animal = {
    g(lst)
}
```

5.2. Implicits and extensions

In this section we attempt to implement a basic SAT solver for Boolean variables. We begin by creating a structure capable of representing boolean expressions that also contain variables, and then work on the solver itself. For this section, work in a single sheet, as you will go back and make slight adjustments along the way.

5.2.1. Implement the type `BoolExpr`, which contains the following constructors:

- `True` and `False`
- `Symbol(s)` - takes a String parameter for the name
- `And(e1, e2)` and `Or(e1, e2)` - takes 2 boolean expressions
- `Not(e)` - takes 1 boolean expression

```
trait BoolExpr


// A case class/object for each required expression subtype
```

5.2.2. Add an implicit method which allows creating BoolExpr values from the basic “true” and “false” Boolean type values.

```
implicit def boolToExpr(b: Boolean): BoolExpr = ???
```

If we want to create expressions at this point, we'd be required to write something like this:

```
And(Not(Or(Symbol("x"), Symbol("y"))), And(True, Symbol("z")))
```

Which is very cumbersome. We would like to use a more convenient and standard notation and usage of operators.

5.2.3. Implement an extension that allows you to use `&&`, `||` and `~` (Unary not) in order to create expressions more easily.

```
extension (e: BoolExpr) {
    ???
}
```

5.2.4. Implement a method in trait BoolExpr, which returns a list of variables in the expression.

```
trait BoolExpr {
  def getVars: List[String]
}


// Override the function definition in each subclass
```

Moving forward, we want to be able to evaluate our expressions built through our implemented features. The only BoolExpr that is not reducible through basic boolean algebraic properties is the `Symbol` type. To simplify a `Symbol` into a `True` or `False` value, we need to give it such a value in a simulated memory. This simulated memory will be a `Map[String, Boolean]` type, taking a name and mapping it's value, to be used later.

Maps are collections of **(key, value)** pairs. Keys should be unique, and every key is associated with a value.
Some of the fundamental operations on maps include:

- retrieving / updating the value associated with a key
- adding a **(key, value)** pair
- removing a **(key, value)** pair

You can find more information on maps on the Scala Docs (https://docs.scala-lang.org/overviews/collections/maps.html).

maps are **immutable**, functions working with maps return a new updated version instead of modifing the map

Some examples with the most used functions can be found below.

Click to hide ⇱

```
let map = Map(1 -> 2, 3 -> 4): Map[Int, Int]
```

- Adding a (key, value) pair to a map

```
map + (5 -> 6)  // Map(1 -> 2, 3 -> 4, 5 -> 6)
map + (3 -> 5)  // Map(1 -> 2, 3 -> 5)  -- if key exists, it updates the value
```

- Removing the pair associated with a key

```
map - (3 -> 4)  // Map(1 -> 2)
```

- Querying a value

```
map get 1  // return 2
map get 3  // return 4
map getOrElse (1, 0)  // return 2
map getOrElse (5, 0)  // return 0 (if key doesn't exist, return provided value)
map contains 1  // True
map contains 5  // False
```

- Higher-order functions

```
map mapValues (x => x + 5)  // Map(1 -> 7, 2 -> 9)
map filterKeys (x => x <= 2)  // Map(1 -> 2)
```

5.2.5. Implement another method in trait BoolExpr, that evaluates the expression, with respect to the stored values given.

```
trait BoolExpr {
```

```
  def getVars: List[String]
  def eval(store: Map[String,Boolean]): Boolean
}

// Override the function definition in each subtype
```

5.2.6. Create class `Solver`, which takes a boolean expression and implement it's 2 member methods:

- `interpretations: List[Store]` - generates all possible memory possibilities for the given expression
- `solve: Option[Store]` - iterates over all stores and returns the first one that satisfies the expression

```
class Solver(formula: BoolExpr){
  type Store = Map[String,Boolean]
  def interpretations: List[Store] = ???
  def solve: Option[Store] = ???
}
```