Functional Programming Midterm Exam

Ada Lovelace, 111999, CO 017, Seat 42

Wednesday 10 November 2021

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 cannot obtain full points.

Stay Functional You are strictly forbidden to use return statements, mutable state (vars) and mutable collections in your solutions.

Some Help The last page of this exam contains an appendix which is useful for formulating your solutions. You can detach this page and keep it aside.

Exercise	**Points**	**Points Achieved**
1	15	
2	15	
3	15	
4	15	
5	15	
Total	75	

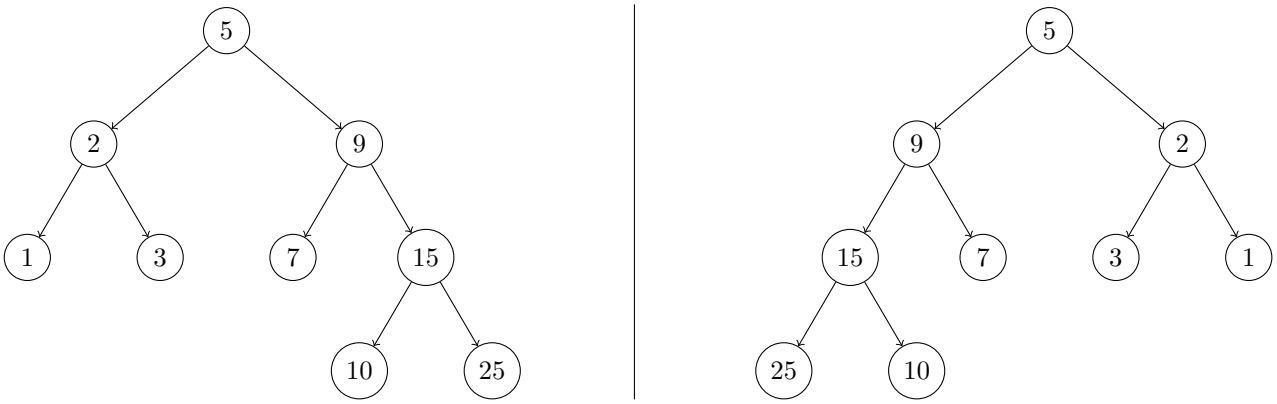
Exercise 1: Recursion (15 points)

In this exercise you will be working with the binary tree structure that you saw during the exercise sessions, defined as follows:

```
trait Tree[T]
case class EmptyTree[T](leq: (T, T) => Boolean) extends Tree[T]
case class Node[T](left: Tree[T], elem: T, right: Tree[T], leq: (T, T) => Boolean)
     extends Tree[T]
```

Your task will be to implement several methods for binary trees. Your implementations must be written directly in the body of `trait Tree`, as opposed to being in the body of the case classes `EmptyTree` and `Node`.

Here are two examples of binary trees:



Exercise 1.1 (7 points)

Implement the `height` method, which returns the height of the tree. The height of a binary tree is defined as the maximum distance between the root node and any leaf node of the tree.

The example trees above have height 4, since the longest path from the root to any leaf contains 4 nodes.

```
trait Tree[T]:
  def height: Int = {




  }
```

Exercise 1.2 (6 points)

Implement the `isBalanced` function, which returns whether the binary tree is balanced. We define a binary tree to be balanced when *all* of the following properties hold:

- An empty binary tree is balanced.
- For a non-empty binary tree, the left sub-tree is balanced.
- For a non-empty binary tree, the right sub-tree is balanced.
- For a non-empty binary tree, the (absolute) difference between the height of the left and the height of the right sub-trees is at most 1.

Both of the example trees shown above are balanced.

Hint: You may use the `height` function that you defined in the exercise 1.2 (even if you didn't complete the exercise).

```
trait Tree[T]:
  def isBalanced: Boolean = {









  }
```

Exercise 2: Pattern Matching (15 points)

Let us use the following data type to represent simple algebraic expressions:

```
enum Expr:
  case Var(name: String)
  case Op(name: String, args: List[Expr])
```

Complete the function `eval(expr: Expr, env: Map[Var, Int]): Int` on the next page so that it evaluates the given expression `expr` with respect to the environement `env` (a mapping from variables to their values).

Your function should only handle the operations with names `"+"` (integer addition) and `"*"` (integer multiplication). If the operation has any other name, you should **`throw`** `UnknownOpException(name)` where `name` must be the name of the unknown operation.

Note that contrary to the exercise that we did in class, here operations can take a variable number of arguments. If a list of arguments is empty, your function should return `0` for addition or `1` for multiplication. If there is a single argument, it should be returned as-is.

If the expression refers to a variable with no mapping in `env`, then your function should **`throw`** `UnknownVarException(name)` where `name` must be the name of the unkown variable.

Example succcessful runs:

```
eval(Op("+", List()), Map())
// == 0

eval(Op("+", List(Var("x"))), Map(Var("x") -> 2))
// == 2

eval(Op("+", List(Var("x"), Var("y"))), Map(Var("x") -> 2, Var("y") -> 3))
// == 5

eval(Op("*", List()), Map())
// == 1

eval(Op("*", List(Var("x"))), Map(Var("x") -> 2))
// == 2

eval(Op("*", List(Var("x"), Var("y"))), Map(Var("x") -> 2, Var("y") -> 3))
// == 6

eval(Op("*", List(Op("+", List(Var("x"), Var("x"))), Var("y"))),
     Map(Var("x") -> 2, Var("y") -> 3))
// == 12
```

Example failing runs:

```
eval(Op("%", List()), Map())
// throws UnknownOpException("%")

eval(Op("+", List(Var("a"), Var("b"))), Map(Var("a") -> 2))
// throws UnknownVarException("b")
```

```
import Expr._
def eval(expr: Expr, env: Map[Var, Int]): Int = {









}
```

Exercise 3: Subtyping and Variance (15 points)

Exercise 3.1 – Variance (6 points)

Consider the following trait:

```
trait Transform[-A, +B]:
  def apply(x: A): B
  def map[C](f: A => C): Transform[A, C]
  def followedBy[C](t: Transform[B, C]): Transform[A, C]
```

It contains an error, where a type parameter is not used at its correct variance.

- Which parameter is it and in which method does it occur? (3pt)
- Can you come up with a modified method signature that is variance correct? (3pt)

Exercise 3.2 – Subtyping (9 points)

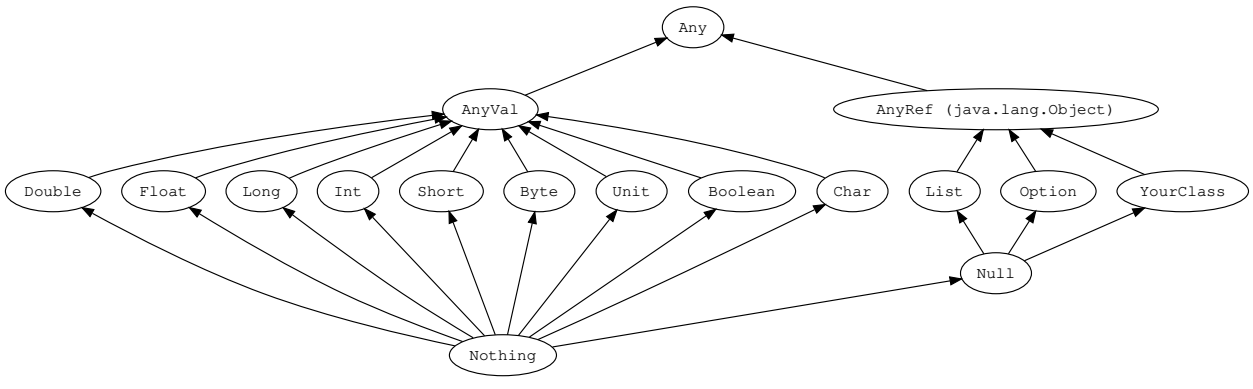
Now also consider the following definition:

```
enum Shape:
  case Point(x: Double, y: Double)
  case Triangle(n1: Point, n2: Point, n3: Point)
  case Circle(center: Point, radius: Double)
```

Which of the subtype assertions are true? (1,5 points per correct answer, 1 point deduction per false answer)

Is the following subtype assertion **true**?			yes	no
`List[Triangle]`	`<:`	`List[AnyRef]`	[ ]	[ ]
`List[Shape]`	`<:`	`List[AnyVal]`	[ ]	[ ]
`List[String]`	`<:`	`List[Shape]`	[ ]	[ ]
`Transform[Point, Circle]`	`<:`	`Transform[Point, Any]`	[ ]	[ ]
`Transform[Point, Shape]`	`<:`	`Transform[Shape, Point]`	[ ]	[ ]
`Transform[Shape, Point]`	`<:`	`Transform[Point, Shape]`	[ ]	[ ]

For reference, here is the hierarchy of Scala types:



There is an arrow from two types `A` to `B` if `A <: B`.

Exercise 4: Structural Induction (15 points)

In this exercise, you will be working with the standard List data structure. Your goal is to prove that the following equivalence holds for all `l1` and `l2` of type `List[A]` and any function `f` of type `A => B`:
`(l1 ++ l2).reverse.map(f) === l2.reverse.map(f) ++ l1.reverse.map(f)`

Axioms. You may use the following axioms:

(1) `Nil.reverse === Nil`

(2) `(x :: xs).reverse === xs.reverse ++ (x :: Nil)`

(3) `(xs ++ ys)++ zs === xs ++ (ys ++ zs)`

(4) `Nil.map(f)=== Nil`

(5) `(x :: xs).map(f)=== f(x):: xs.map(f)`

(6) `Nil ++ xs === xs`

(7) `xs ++ Nil === xs`

(8) `(x :: xs)++ ys === x :: (xs ++ ys)`

(9) `(l1 ++ l2).map(f)=== l1.map(f)++ l2.map(f)`

Exercise 4.1 (10 points)

Prove the following theorem:

(10) `(l1 ++ l2).reverse === l2.reverse ++ l1.reverse`

Exercise 4.2 (5 points)

Prove that: `(l1 ++ l2).reverse.map(f)=== l2.reverse.map(f)++ l1.reverse.map(f)`

Hint: In question 1.2, you may use axiom (10), even if you do not manage to solve exercise 4.1.

Note: Be very precise in your proof:

- Clearly state which axiom, lemma or hypothesis you use at each step.
- Use only 1 axiom, lemma or hypothesis at each step.
- Underline the part of an equation on which you apply your axiom, lemma or hypothesis.
- Make sure to state what you want to prove and what your induction hypotheses are, if any.

Exercise 5: Lists (15 points)

In this exercise, you will be working with the standard `List` data structure. The goal is to split a list of integers into strictly increasing sub-lists. The parts of this exercise can be solved independently. Even if you did not solve a question, you can use the function assuming it is correctly implemented.

Exercise 5.1 (7 points)

Implement the function `takeWhileStrictlyIncreasing` that, given a list of integers `list`, returns the longest strictly increasing prefix.

Some example calls:

```
takeWhileStrictlyIncreasing(List(1, 8, 9, 9, 10, 2, 3))
// == List(1, 8, 9)

takeWhileStrictlyIncreasing(List(9, 10, 1, 2, 3, 11, 12, 13))
// == List(9, 10)

takeWhileStrictlyIncreasing(List())
// == List()
```

```
def takeWhileStrictlyIncreasing(list: List[Int]): List[Int] = {









}
```

Exercise 5.2 (8 points)

Implement the function `increasingSequences` that, given a list of integers `list`, returns the strictly increasing contiguous sub-lists. You must use the function `takeWhileStrictlyIncreasing` defined above. You can assume it is correctly implemented even if you did not solve the previous question.

Some example calls:

```
increasingSequences(List(8, 9, 1, 2, 3, 3, 4))
// == List(List(8, 9), List(1, 2, 3), List(3, 4))

increasingSequences(List(1))
// == List(List(1))

increasingSequences(List())
// == List()
```

```
def increasingSequences(list: List[Int]): List[List[Int]] = {









}
```

Appendix: Scala Standard Library Methods

Here are some methods from the Scala standard library that you may find useful, on `List[A]`:

- `xs.head: A`: returns the first element of the list. Throws an exception if the list is empty.
- `xs.tail: List[A]`: returns the list `xs` without its first element. Throws an exception if the list is empty.
- `x :: (xs: List[A]): List[A]`: prepends the element `x` to the left of `xs`, returning a `List[A]`.
- `xs ++ (ys: List[A]): List[A]`: appends the list `ys` to the right of `xs`, returning a `List[A]`.
- `xs.apply(n: Int): A`, or `xs(n: Int): A`: returns the `n`-th element of `xs`. Throws an exception if there is no element at that index.
- `xs.drop(n: Int): List[A]`: returns a `List[A]` that contains all elements of `xs` except the first `n` ones. If there are less than `n` elements in `xs`, returns the empty list.
- `xs.filter(p: A => Boolean): List[A]`: returns all elements from `xs` that satisfy the predicate `p` as a `List[A]`.
- `xs.flatMap[B](f: A => List[B]): List[B]`: applies `f` to every element of the list `xs`, and flattens the result into a `List[B]`.
- `xs.foldLeft[B](z: B)(op: (B, A) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going left to right.
- `xs.foldRight[B](z: B)(op: (A, B) => B): B`: applies the binary operator `op` to a start value and all elements of the list, going right to left.
- `xs.map[B](f: A => B): List[B]`: applies `f` to every element of the list `xs` and returns a new list of type `List[B]`.
- `xs.nonEmpty: Boolean`: returns **`true`** if the list has at least one element, **`false`** otherwise.
- `xs.reverse: List[A]`: reverses the elements of the list `xs`.
- `xs.take(n: Int): List[A]`: returns a `List[A]` containing the first `n` elements of `xs`. If there are less than `n` elements in `xs`, returns these elements.
- `xs.size: Int`: returns the number of elements in the list.
- `xs.zip(ys: List[B]): List[(A, B)]`: zips elements of `xs` and `ys` in a pairwise fashion. If one list is longer than the other one, remaining elements are discalaarded. Returns a `List[(A, B)]`.
- `xs.toSet: Set[A]`: returns a set of type `Set[A]` that contains all elements from the list `xs`. Note that the resulting set will contain no duplicates and may therefore be smaller than the original list.