EPFL

1

Profs. Martin Odersky and Viktor Kuncak
CS-210 Functional Programming
09.11.2022 from 13h15 to 14h45
Duration : 90 minutes

SCIPER : **1000001** ROOM : **CO1**

Ada Lovelace

Wait for the start of the exam before turning to the next page. This document is printed double sided, 16 pages. Do not unstaple.

Material	This is a closed book exam. Paper documents and electronic devices are not allowed. Place on your desk your student ID and writing utensils. Place all other personal items below your desk or on the side. If you need additional draft paper, raise your hand and we will provide some.
Time	All points are not equal: we do not think that all exercises have the same difficulty, even if they have the same number of points.
Appendix	The last page of this exam contains an appendix which is useful for formulating your solutions. Do not detach this sheet.
Use a pen	For technical reasons, **only use black or blue pens for the MCQ part, no pencils!** Use white corrector if necessary.
Stay Functional	You are strictly forbidden to use `return` statements, mutable state (`vars`) and mutable collections in your solutions.

First part: single choice questions

Each question has ***exactly one*** *correct answer. Marking only the box corresponding to the correct answer will get you 4 points. Otherwise, you will get 0 points for the question.*

Given the following function `sums`:

```
def add(c: Int, acc: List[(Int, Int)]): List[(Int, Int)] = acc match
  case Nil => List((c, 1))
  case x :: xs => if x._1 == c then (c, x._2+1) :: xs else x :: add(c, xs)

def sums(digits: List[Int]): List[(Int, Int)] =
  digits.foldRight(List[(Int, Int)]())(add)
```

Your task is to identify several operations on lists of digits:

Question [`SCQ-01`] What does the following operation implement, for a given input list of digits?

```
def mystery1(digits: List[Int]): List[Int] =
  sums(digits).filter(_._2 == 1).map(_._1)
```

- [x] Returns a list of elements that appear exactly once in the input list, in a reverse order
- [ ] Returns a list of elements that appear exactly once in the input list, preserving the original order of appearance
- [ ] Returns a list consisting of elements of the input list that are equal to `1`
- [ ] Returns the first element of the input list
- [ ] Returns `List(1)` if the input list contains at least one digit `1`, an empty list otherwise

Question [`SCQ-02`] What does the following operation implement, for a given input list of digits?

```
def mystery2(digits: List[Int]): List[Int] =
  mystery1(digits).filter(_ == 1)
```

- [x] Returns `List(1)` if the input list contains exactly one digit `1`, an empty list otherwise
- [ ] Returns `List(1)` if the input list contains at least one digit `1`, an empty list otherwise
- [ ] Returns a list of elements that appear exactly once in the input list, preserving the original order of appearance
- [ ] Returns a list of elements that appear exactly once in the input list, in a reverse order
- [ ] Returns a list consisting of elements of the input list that are equal to `1`

Question [SCQ-03] What does the following operation implement, for a given input list of digits?

```
def mystery3(digits: List[Int]): Int = sums(digits) match
  case Nil => 0
  case t => t.reduceLeft((a, b) => (a._1 * a._2 + b._1 * b._2, 1))._1
```

- [x] Returns the sum of all elements in the input list
- [ ] Returns the sum of elements that appear exactly once in the input list
- [ ] Returns the sum of elements in the input list, counting duplicated elements only once
- [ ] Returns the product of all elements in the input list
- [ ] Returns the number of elements in the input list

Question [SCQ-04] What does the following operation implement, for a given input list of digits?

```
def mystery4(digits: List[Int]): Int = sums(digits) match
  case Nil => 0
  case t => t.reduceLeft((a, b) => (a._1, a._2 + b._2))._2
```

- [x] Returns the number of elements in the input list
- [ ] Returns the number of elements in the input list, counting duplicated elements only once
- [ ] Returns the number of elements that appear exactly once in the input list
- [ ] Returns the sum of all elements in the input list
- [ ] Returns the first element of the input list

The goal of the 4 following questions is to prove that the methods `map` and `mapTr` are equivalent. The former is the version seen in class and is specified by the lemmas `MapNil` and `MapCons`. The later version is a tail-recursive version and is specified by the lemmas `MapTrNil` and `MapTrCons`.

All lemmas on this page hold for all `x: Int`, `y: Int`, `xs: List[Int]`, `ys: List[Int]`, `l: List [Int]` and `f: Int => Int`.

Given the following lemmas:

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

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

(MapTrNil) `Nil.mapTr(f, ys) === ys`

(MapTrCons) `(x :: xs).mapTr(f, ys) === xs.mapTr(f, ys ++ (f(x) :: Nil))`

(NilAppend) `Nil ++ xs === xs`

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

Let us first prove the following lemma:

(AccOut) `l.mapTr(f, y :: ys) === y :: l.mapTr(f, ys)`

We prove it by induction on `l`.

Question [SCQ-05] *Base case:* `l` is `Nil`. Therefore, we need to prove:

`Nil.mapTr(f, y :: ys) === y :: Nil.mapTr(f, ys).`

What exact sequence of lemmas should we apply to rewrite the left hand-side (`Nil.mapTr(f, y :: ys)`) to the right hand-side (`y :: Nil.mapTr(f, ys)`)?

- [ ] NilAppend, NilAppend, MapTrNil
- [ ] NilAppend, NilAppend, NilAppend
- [x] MapTrNil, MapTrNil
- [ ] NilAppend, MapTrNil, NilAppend
- [ ] MapTrNil, NilAppend
- [ ] NilAppend, MapTrNil

Question [SCQ-06] *Induction step:* `l` is `x :: xs`. Therefore, we need to prove:

`(x :: xs).mapTr(f, y :: ys) === y :: (x :: xs).mapTr(f, ys)`

We name the induction hypothesis IH.

What exact sequence of lemmas should we apply to rewrite the left hand-side (`(x :: xs).mapTr(f, y :: ys)`) to the right hand-side (`y :: (x :: xs).mapTr(f, ys)`)?

- [x] MapTrCons, ConsAppend, IH, MapTrCons
- [ ] NilAppend, ConsAppend, IH, MapTrCons
- [ ] NilAppend, IH, MapTrCons
- [ ] ConsAppend, MapTrCons, IH
- [ ] ConsAppend, IH, MapTrCons
- [ ] MapTrCons, IH, ConsAppend, MapTrCons
- [ ] IH, ConsAppend, IH, ConsAppend
- [ ] MapTrCons, NilAppend, IH, MapTrCons
- [ ] NilAppend, ConsAppend, IH, ConsAppend

Given all lemmas on the previous page, including AccOut, let us now prove our goal:

(MapEqMapTr) `l.map(f) === l.mapTr(f, Nil)`

We prove it by induction on `l`.

Question [SCQ-07] *Base case:* `l` is `Nil`. Therefore, we need to prove:

`Nil.map(f) === Nil.mapTr(f, Nil)`

What exact sequence of lemmas should we apply to rewrite the left hand-side (`Nil.map(f)`) to the right hand-side (`Nil.mapTr(f, Nil)`)?

- [ ] MapTrNil, MapTrNil
- [ ] MapTrNil, NilAppend
- [ ] NilAppend, MapTrNil
- [ ] MapNil, MapNil
- [ ] MapNil, NilAppend
- [ ] NilAppend, MapNil
- [ ] MapTrNil, MapNil
- [x] MapNil, MapTrNil

Question [SCQ-08] *Induction step:* `l` is `x :: xs`. Therefore, we need to prove:

`(x :: xs).map(f) === (x :: xs).mapTr(f, Nil)`

We name the inductions hypothesis IH.

What exact sequence of lemmas should we apply to rewrite the left hand-side (`(x :: xs).map(f)`) to the right hand-side (`(x :: xs).mapTr(f, Nil)`)?

- [ ] MapCons, NilAppend, IH, AccOut, MapTrCons
- [ ] MapCons, IH, NilAppend, AccOut, MapTrCons
- [ ] MapCons, NilAppend, IH, AccOut, MapTrCons
- [ ] MapCons, AccOut, IH, NilAppend, MapTrCons
- [x] MapCons, IH, AccOut, NilAppend, MapTrCons
- [ ] MapCons, NilAppend, AccOut, IH, MapTrCons
- [ ] MapCons, IH, NilAppend, MapTrCons, AccOut
- [ ] MapCons, NilAppend, AccOut, MapTrCons, IH
- [ ] MapCons, IH, IH, NilAppend, MapTrCons
- [ ] MapCons, NilAppend, AccOut, AccOut, MapTrCons
- [ ] MapCons, IH, NilAppend, MapTrCons, IH
- [ ] MapCons, NilAppend, AccOut, MapTrCons, AccOut
- [ ] MapTrCons, IH, NilAppend, AccOut, MapCons
- [ ] MapTrCons, IH, AccOut, NilAppend, MapCons
- [ ] MapTrCons, AccOut, NilAppend, IH, MapCons
- [ ] MapTrCons, NilAppend, IH, IH, MapCons

Note: question 8 is graded independently from questions 5 and 6; you can use the AccOut lemma as an axiom even if you did not prove it.

Catalog

Given the following classes:

- **class** `Pair[+U, +V]`
- **class** `Iterable[+U]`
- **class** `Map[U, +V]` **extends** `Iterable[Pair[U, V]]`

Recall that + means covariance, – means contravariance and no annotation means invariance (i.e. neither covariance nor contravariance).

Consider also the following typing relationships for `A`, `B`, `X`, and `Y`:

- `A >: B`
- `X >: Y`

Fill in the subtyping relation between the types below using symbols:

- `<:` in case `T1` is a subtype of `T2`;
- `>:` in case `T1` is a supertype of `T2`;
- "Neither" in case `T1` is neither a supertype nor a supertype of `T2`.

Question [SCQ-09] What is the correct subtyping relationship between `A => (Y => X)` and `A => (X => Y)`?

- [ ] `<:`
- [x] `>:`
- [ ] Neither

Question [SCQ-10] What is the correct subtyping relationship between `Map[A, X]` and `Map[B, Y]`?

- [ ] `<:`
- [ ] `>:`
- [x] Neither

Question [SCQ-11] What is the correct subtyping relationship between `Iterable[Pair[A, Y]] => Y` and `Map[A, Y] => X`?

- [x] `<:`
- [ ] `>:`
- [ ] Neither

Question [SCQ-12] What does the following operation output for a given input list of numbers ?

```
def mystery5(ys: List[Int]) =
  for y <- ys if y >= 0 && y <= 255 yield
    val bits =
      for z <- 7 to 0 by -1 yield
        if ((1 << z) & y) != 0 then "1" else "0"
    bits.foldRight("")((z, acc) => z + acc)
```

We have as an output...

- [x] ... a list of strings, each string corresponding to the 8-bit representation of an element if and only if that element is between 0 and 255 included. The most significant bit is farthest to the left in the characters sequence.
- [ ] ... a list of strings, each string corresponding to the 8-bit representation of an element if and only if that element is between 0 and 255 included. The most significant bit is farthest to the right in the characters sequence.
- [ ] ... a list of lists of elements ∈ {"0","1"}, each list corresponding to the 8-bit representation of an element of the input list if and only if that element is between 0 and 255 included. The most significant bit is farthest to the right in the string sequence.
- [ ] ... a list of lists of elements ∈ {"0","1"}, each list corresponding to the 8-bit representation of an element of the input list if and only if that element is between 0 and 255 included. The most significant bit is farthest to the left in the string sequence.

Hint: The most significant bit represents the largest value in a multiple-bit binary number.

Question [SCQ-13] Given the following method:

```
def mystery6(nIter: Int) (ss: List[String] ): List[String] =
  if nIter <= 0 then ss
  else mystery6 (nIter - 1) (
    for
      s <- ss
      c <- List ('c' , 'b' , 'a')
    yield
      s + c
  ) ::: ss
```

What is the output if we call `mystery6` this way:
`mystery6(5)(List("")).filter(_.exists(_ == 'b'))(0)`

- [x] `"ccccb"`
- [ ] `"bcccc"`
- [ ] `"acccb"`
- [ ] `"b"`
- [ ] `"abc"`
- [ ] `"ab"`
- [ ] `"bccca"`

Second part, open questions

Question 14 *This question is worth 13 points.*

A reflected binary code or simply Gray code is an n-bit binary encoding C_n. It has the property that successive codewords only differ by a single bit, *i.e.*, $D(C_n(i), C_n(i+1)) = 1$, where $D(x, y)$ denotes the Hamming distance between x and y for $i \in \mathbb{Z}_{2^n}$. Gray codes for a few small n are given in the codebox below.

In this exercise, we wish to construct Gray codes of arbitrary sizes in a **succinct** and **recursive** manner. This means that your program **must not** exceed **10** lines of code and **must not** contain any helper functions. Note that the order of elements in the obtained lists is of crucial importance and needs to **exactly** correspond to the given examples. A tail-recursive solution **is not** required.

Here are some examples of successful runs:

```
gray(0)
// : List[String] = List("")

gray(1)
// : List[String] = List("0", "1")

gray(2)
// : List[String] = List("00", "01", "11", "10")

gray(3)
// : List[String]
// = List(
//     "000", "001", "011", "010",
//     "110", "111", "101", "100"
// )
```

Hint 1: Proceed in an inductive manner, in other words, assuming that C_{n-1} is a valid Gray code what transformation $F(C_{n-1}) = C_n$ yields a correct encoding for n?
Hint 2: Use the + operator to build strings.

☐0 ☐1 ☐2 ☐3 ☐4 ☐5 ☐6
☐7 ☐8 ☐9 ☐10 ☐11 ☐12 ■13 *Do not write here.*

```
// Returns the gray code of size n.
def gray(n: Int): List[String] =
  require(n >= 0)
```

Question 15 *This question is worth 35 points. It contains 5 sub-questions of 7 points each.*

In this exercise, you will implement the k-nearest neighbors algorithm (k-NN), in which you will use a training dataset in order to classify new test points. The algorithm works by first finding the k nearest training points for a given test point, and then assigning the label of the majority class of those k points to that test point.

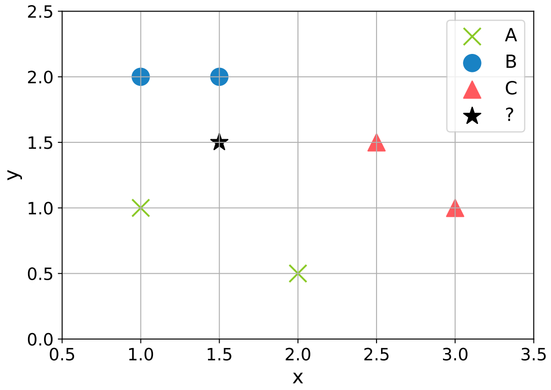

Figure 1: k-NN example: To classify the given *test* point $\star$, we simply assign it the majority label of the k nearest *training* points. For example, the $k = 3$ nearest neighbors have class labels A, B, B, so we label the new point as B.

We give you the following definitions of a 2D point, with or without a label, and a function to compute the distance of a test point to a list of training points.

```
import scala.math.sqrt

// A point in 2D space
case class Point(x: Double, y: Double):
  // Euclidean distance to another point
  def distance(p: Point): Double =
    val dx = x - p.x
    val dy = y - p.y
    sqrt(dx * dx + dy * dy)

// A 2D point with a label
case class LabeledPoint(point: Point, label: String)

// Computes the distance of a point p to each point in the list pts.
def distances(pts: List[LabeledPoint], p: Point): List[(LabeledPoint, Double)] =
  pts.map(q => (q, p.distance(q.point)))
```

Here is the interface of the functions you should implement:

```
// Retrieves the k nearest training points for a given point p.
def kNearest(pts: List[LabeledPoint], p: Point, k: Int): List[LabeledPoint]

// Given a list of labeled points, counts the number of times each label appears.
def countLabels(pts: List[LabeledPoint]): Map[String, Int]

// Given the label count, returns the label with the highest count. If multiple
// labels have the same maximum count, returning either of them is fine.
def maxLabel(labelCount: Map[String, Int]): String

// Given a list of labeled points, returns the majority label.
def majorityLabel(pts: List[LabeledPoint]): String

// k-NN function that labels a list of test points using the k nearest neighbors
// in the training set.
def knn(train: List[LabeledPoint], test: List[Point], k: Int): List[LabeledPoint]
```

Here is an example of a successful run:

```
// A set of labeled training points
val train = List(
  LabeledPoint(Point(1.0, 1.0), "A"), LabeledPoint(Point(2.0, 0.5), "A"),
  LabeledPoint(Point(1.0, 2.0), "B"), LabeledPoint(Point(1.5, 2.0), "B"),
  LabeledPoint(Point(3.0, 1.0), "C"), LabeledPoint(Point(2.5, 1.5), "C"),
)
// A set of test points to classify
val test = List(Point(1.5, 1.5), Point(1.5, 1.0))

knn(train, test, 3)
// : List[LabeledPoint]
// = List(
//     LabeledPoint(Point(1.5, 1.5), "B"),
//     LabeledPoint(Point(1.5, 1.0), "A")
// )
```

Implement all the functions indicated in the interface above. It is possible to solve all of them using a single line of code each.
Hint: Remember that some of the functions we ask you to implement can be helpful in implementing others. It's okay to use these functions even if you did not solve them first.

0 1 2 3 4 5 6 7 *Do not write here.*

```
// Retrieves the k nearest training points for a given point p.
def kNearest(pts: List[LabeledPoint], p: Point, k: Int): List[LabeledPoint] =
```

0 1 2 3 4 5 6 7 *Do not write here.*

```
// Given a list of labeled points, counts the number of times each label appears.
def countLabels(pts: List[LabeledPoint]): Map[String, Int] =
```

0 1 2 3 4 5 6 7 *Do not write here.*

```
// Given the label count, returns the label with the highest count. If multiple
// labels have the same maximum count, returning either of them is fine.
def maxLabel(labelCount: Map[String, Int]): String =
```

0 1 2 3 4 5 6 7 *Do not write here.*

```
// Given a list of labeled points, returns the majority label.
def majorityLabel(pts: List[LabeledPoint]): String =
```

0 1 2 3 4 5 6 7 *Do not write here.*

```
// k-NN function that labels a list of test points using the k nearest neighbors
// in the training set.
def knn(train: List[LabeledPoint], test: List[Point], k: Int): List[LabeledPoint] =
```

CATALOG

Appendix: Scala Standard Library Methods

Here are the prototypes of some Scala classes that you might find useful:

```
abstract class List[+A]:
  // Adds an element at the beginning of this list.
  def ::[B >: A](elem: B): List[B]
  // A copy of this sequence with an element appended.
  def appended[B >: A](elem: B): List[B]
  // Get the element at the specified index.
  def apply(n: Int): A
  // Selects all elements except first n ones.
  def drop(n: Int): Iterable[A]
  // Selects all elements of this list which satisfy a predicate.
  def filter(pred: (A) => Boolean): List[A]
  // Applies a binary operator to a start value and all elements of this sequence,
    going left to right.
  def foldLeft[B](z: B)(op: (B, A) => B): B
  // Applies a binary operator to a start value and all elements of this sequence,
    going right to left.
  def foldRight[B](z: B)(op: (A, B) => B): B
  // Partitions the list into a map of lists according to some discriminator function.
  def groupBy[K](f: (A) => K): Map[K, List[A]]
  // Selects the first element of this list.
  def head: A
  // Selects the last element.
  def last: A
  // Applies the function f to each element in the list.
  def map[B](f: (A) => B): List[B]
  // The size of this collection.
  def size: Int
  // Sorts this sequence according to the Ordering which results from transforming an
    implicitly given Ordering with a transformation function.
  def sortBy[B](f: (A) => B): List[A]
  // Sorts this sequence according to a comparison function.
  def sortWith(lt: (A, A) => Boolean): List[A]
  // Selects all elements except the first.
  def tail: List[A]
  // Selects the first n elements.
  def take(n: Int): Iterable[A]
abstract class Map[K, +V]:
  // Optionally returns the value associated with a key.
  def get(key: K): Option[V]
  // Builds a new map by applying a function to all elements of this map.
  def map[K2, V2](f: ((K, V)) => (K2, V2)): Map[K2, V2]
  // Finds the first element which yields the largest value measured by function f.
  def maxBy[B](f: ((K, V)) => B): (K, V)
  // Returns this map as a List[(K, V)].
  def toList: List[(K, V)]
abstract final class Int:
  // Returns this value bit-shifted left by the specified number of bits, filling in
  // the new right bits with zeroes.
  def <<(x: Int): Long
abstract final class Long:
  // Returns the bitwise AND of this value and x.
  def &(x: Int): Long
abstract final class String:
  // Tests whether a predicate holds for at least one element of this sequence.
  def exists(p: Char => Boolean): Boolean
```