DS 644: Spring 2023 **Midterm Examination** **Section 104**

Cheating will not be tolerated. If there is any indication that a student may have given or received unauthorized aid on this test, the case will be referred to the Office of the Dean of Students. When you finish the exam, you must sign the following pledge:

"On my honor as a student I, _____, have neither given nor received unauthorized aid on this exam." (print name clearly)

Signature: _____ Date: 29 March 2023

Page:	2	3	5	6	7	9	Total
Points:	13	27	6	15	18	21	100
Score:							

Instructions. Answer the following multiple choice questions by selecting all correct choices. If a question has more than one correct choice, it will say in parentheses how many items you should select. *Select all correct choices to receive full credit!*

1. (6 pts) **Programming Paradigms**

   (a) Which of the following is *not* an example of a programming paradigm?

   ☐ Declarative ☐ Functional ☐ Immutable ☐ Imperative ☐ Object-oriented

   (b) Which of the following characteristics are typical of imperative programs? (select two)

   ☐ Values of variables do not change or "mutate" (they are *immutable*).
   ☐ Iteration is typically performed with `for` or `do-while` loops.
   ☐ Iteration is typically performed with recursion.
   ☐ Functions often have *side-effects*.

   (c) Which of the following characteristics are typical of functional programs? (select two)

   ☐ Values of variables do not change or "mutate" (they are *immutable*).
   ☐ Iteration Is typically performed with `for` or `do-while` loops.
   ☐ Iteration is typically performed with recursion.
   ☐ Functions often have *side-effects*.

2. (2 pts) A *higher-order function* is a function that

   ☐ can be passed as an argument to other functions.
   ☐ can be returned as output by other functions.
   ☐ can be called a higher order of times than ordinary functions.
   ☐ accepts another function as input.
   ☐ takes a long time to compute (i.e., has higher order time complexity).

3. (2 pts) Which *two* of the following terms are used to describe the programming paradigm of Scala?

   ☐ assembly ☐ declarative ☐ functional ☐ imperative ☐ object-oriented

4. (3 pts) What is the result of the following program?

```
val x = 2
def f(y: Int) = y * y
val result = {
  val x = f(2)
  x * x
} + x
```

   ☐ 6 ☐ 16 ☐ 18 ☐ 32 ☐ None—it does not terminate.

5. (3 pts) Why do you think the designers of Scala thought it was important to support "tail-recursive" functions?

   ☐ Because recursion should be carried out on the tail, not the head.

   ☐ Because recursion should be carried out on the head, not the tail.

   ☐ Because they are "stack-safe"—they help us avoid stack overflows.

   ☐ Because they are "disk-safe"—they help us avoid network storage leaks.

6. (12 pts) The parts below refer to the function

```
def test(x:Bool, y:Int) = if (x) (y + 2)/y else 0
```

   Let CBN = call-by-name

   and CBV = call-by-value.

   (a) Which strategy evaluates `test(true, 2)` most efficiently (in the fewest steps)?
   ☐ CBN ☐ CBV ☐ CBN and CBV require the same number of steps

   (b) Which strategy evaluates `test(true, 1+1)` most efficiently?
   ☐ CBN ☐ CBV ☐ CBN and CBV require the same number of steps

   (c) Which strategy evaluates `test(false, 2)` most efficiently?
   ☐ CBV ☐ CBN ☐ CBN and CBV require the same number of steps

   (d) Which strategy evaluates `test(false, 1+1)` most efficiently?
   ☐ CBN ☐ CBV ☐ CBN and CBV require the same number of steps

7. (12 pts) **Higher-order Functions**

   All parts of this question refer to the following `sum` function.

```
def sum(f: Int => Int): (Int, Int) => Int = {
  def sumF(a: Int, b: Int): Int = {
    if (a > b) 0
    else f(a) + sumF(a + 1, b)
  }
  sumF
}
```

   (a) What does `sum(2, 3)` compute?

   ☐ `2 + 3`

   ☐ `2 + 2 + 3 + 3`

   ☐ `2 * 2 + 3 * 3`

   ☐ a function that takes two integer arguments and returns their sum

   ☐ `sum(2, 3)` causes a run-time error.

   ☐ `sum(2, 3)` causes a compile-time error.

(b) What does `sum(x => x)(2, 3)` compute?

- ☐ `2 + 3`
- ☐ `2 + 2 + 3 + 3`
- ☐ `2 * 2 + 3 * 3`
- ☐ a function that takes two integer arguments and returns their sum
- ☐ `sum(x => x)(2, 3)` causes a run-time error.
- ☐ `sum(x => x)(2, 3)` causes a compile-time error.

(c) What does `sum(x => x)` return?

- ☐ `2 + 3`
- ☐ `2 + 2 + 3 + 3`
- ☐ `2 * 2 + 3 * 3`
- ☐ a function that takes two integer arguments and returns their sum
- ☐ `sum(x => x)(2, 3)` causes a run-time error.
- ☐ `sum(x => x)(2, 3)` causes a compile-time error.

(d) What does `sum(x => x + x)(2, 3)` compute?

- ☐ `2 + 3`
- ☐ `2 + 2 + 3 + 3`
- ☐ `2 * 2 + 3 * 3`
- ☐ a function that takes two integer arguments and returns their sum
- ☐ `sum(x => x + x)(2, 3)` causes a run-time error.
- ☐ `sum(x => x + x)(2, 3)` causes a compile-time error.

(e) What does `sum(x => x * x)(2, 3)` compute?

- ☐ `2 + 3`
- ☐ `2 + 2 + 3 + 3`
- ☐ `2 * 2 + 3 * 3`
- ☐ a function that takes two integer arguments and returns their sum
- ☐ `sum(x => x * x)(2, 3)` causes a run-time error.
- ☐ `sum(x => x * x)(2, 3)` causes a compile-time error.

(f) What does `sum(x => x / 1.0)(2, 3)` compute?

- ☐ `2 + 3`
- ☐ `2 + 2 + 3 + 3`
- ☐ `2 * 2 + 3 * 3`
- ☐ a function that takes two integer arguments and returns their sum
- ☐ `sum(x => x/1.0)(2, 3)` causes a run-time error.
- ☐ `sum(x => x/1.0)(2, 3)` causes a compile-time error.

8. (3 pts) Consider the general form of pattern matching in Scala,

```
e match { case p1 => e1 ... case pn => en }
```

Which of the following are true statements?

- ☐ Scala matches the value of the selector `e` with the patterns `p1, ..., pn` in the order in which they are written.
- ☐ The match expression is rewritten to the right-hand side of the first case where the pattern matches the selector `e`.
- ☐ References to pattern variables are replaced by the corresponding parts in the selector.
- ☐ All of the above.
- ☐ None of the above.

9. (3 pts) Consider the following Scala program.

```
trait Expr
case class Number(n: Int) extends Expr
case class Sum(e1: Expr, e2: Expr) extends Expr

object Number{ def apply(n: Int) = new Number(n) }

object Sum{ def apply(e1: Expr, e2: Expr) = new Sum(e1, e2) }

def eval(e: Expr): Int = e match {
  case Number(n) => n
  case Sum(e1, e2) => eval(e1) + eval(e2)
}
```

What is the result of the following expression?

```
eval(Sum(Number(1), Number(2)))
```

☐ 0 ☐ 1 ☐ 2 ☐ 3 ☐ None of the above.

10. (6 pts) Let `val X = List(1, 2, 3)`
and `val Y = List(1, 2, 3)`.

(a) To what does the expression `X.map(x => Y.map(y => y - x))` evaluate?

- ☐ `List(0, 0, 0, 0, 0, 0, 0, 0, 0)`
- ☐ `List(List(0, 0, 0), List(0, 0, 0), List(0, 0, 0))`
- ☐ `List(0, -1, -2, 1, 0, -1, 2, 1, 0)`
- ☐ `List(0, 1, 2, -1, 0, 1, -2, -1, 0)`
- ☐ `List(List(0, 1, 2), List(-1, 0, 1), List(-2, -1, 0))`

(b) To what does the expression `X.flatMap(x => Y.map(y => y - x))` evaluate?

- ☐ `List(0, 0, 0, 0, 0, 0, 0, 0, 0)`
- ☐ `List(List(0, 0, 0), List(0, 0, 0), List(0, 0, 0))`
- ☐ `List(List(0, -1, -2), List(1, 0, -1), List(2, 1, 0))`
- ☐ `List(0, 1, 2, -1, 0, 1, -2, -1, 0)`
- ☐ `List(List(0, 1, 2), List(-1, 0, 1), List(-2, -1, 0))`

11. (9 pts) Reducing lists with `foldLeft`.

Suppose you want to implement a (polymorphic) `reverse` function, which reverses the order of a given list, `xs: List[T]`, using Scala's `foldLeft` function.

You start with

```
def reverse[T](xs: List[T]): List[T] = (xs foldLeft ???)((ys, y) => ???)
```

(a) What aspect of the code above tells you that this `reverse` function will be *polymorphic*?

- ☐ It operates on lists.
- ☐ The second `???` will be a function, so it's "higher-order."
- ☐ There is a folding or "reduction" operation involved.
- ☐ It is recursive.
- ☐ It takes a type parameter `T`.

(b) The first set of three question marks `???` should be replaced with which of the following?

☐ `Nil` ☐ `List[T]()` ☐ `List[T](0)` ☐ `ys :: y` ☐ `y :: ys`

(c) The second set of three question marks `???` should be replaced with which of the following?

☐ `Nil` ☐ `List[T]()` ☐ `List[T](0)` ☐ `ys :: y` ☐ `y :: ys`

12. (18 pts) Consider the abstract class `IntSet`, the (concrete) `Empty` object, and the (concrete, partially implementated) NonEmpty IntSet class, shown below.

```
abstract class IntSet {

    def incl(n: Int): IntSet
    def remove(n: Int): IntSet

    def filter(p: Int => Boolean): IntSet = filterAcc(p, Empty)

    def filterAcc(p: Int => Boolean, acc: IntSet): IntSet = {
        if (p(elem)) this.remove(elem).filterAcc(p, acc incl elem)
        else this.remove(elem).filterAcc(p, acc)
    }
    // ... other methods ...
}

object Empty extends IntSet {
    ↑ incl
    def incl(n: Int): IntSet = new NonEmpty(n, Empty, Empty)
    ↑ remove
    def remove(n: Int): IntSet = throw new java.util.NoSuchElementException

    def union(that: IntSet): IntSet = that
}

class NonEmpty(elem: Int, left: IntSet, right: IntSet) extends IntSet {
    ↑ incl
    def incl(n: Int): IntSet =
        if (n < elem) new NonEmpty(elem, ___(1)___, ___(2)____)

        else if (n > elem) new NonEmpty(elem, ___(3)___, ___(4)___)

        else ___(5)___
    }

    def union(that: IntSet): IntSet = ___(6)___

    def remove(n: Int): IntSet =
        if (n < elem) new NonEmpty(elem, ___(7)___, ___(8)____)

        else if (n > elem) new NonEmpty(elem, ___(9)___, ___(10)___)

        else ___(11)___
    // ... other methods ...
```

(a) If `s` is a `NonEmpty IntSet` representing the set {1, 2, 3, 4, 5}, what does the call `s.filter(x => x < 3)` do?

- ☐ It returns an `Empty IntSet`.
- ☐ It returns a `NonEmpty IntSet` representing the set {1, 2}.
- ☐ It throws a `NoSuchElementException`.
- ☐ Nothing; the function call does not terminate.

(b) Given your answer to the last part, what can you say about the above implementation of `filterAcc`? (select two)

- ☐ It is a recursive function that returns a filtered `IntSet`.
- ☐ It is a non-recursive function that never terminates.
- ☐ It is a recursive function that handles the base case correctly.

☐ It is a recursive function that does not handle the base case correctly.

☐ It should not be implemented in the abstract `IntSet` class.

(c) Assuming `incl(n: Int)` should return a new `IntSet` which contains all elements of `this` set, along with the the new element `n` in case it does not already exist in this set, what goes in spaces (1), (2)?

☐ `left, right` ☐ `left.incl(n), right` ☐ `left, right.incl(n)`

☐ `left.remove(n), right` ☐ `left, right.remove(n)`

(d) What goes in (3), (4)?

☐ `left, right` ☐ `left.incl(n), right` ☐ `left, right.incl(n)`

☐ `left.remove(n), right` ☐ `left, right.remove(n)`

(e) What goes in (5)?

☐ `this`

☐ `this union that`

☐ `left union right`

☐ `(left union right) incl that`

☐ `((left union right) union that) incl n`

(f) Assuming `union(that: IntSet)` should return a new `IntSet` that is the union of the `IntSet`s `this` and `that`, what goes in (6)?

☐ `this`

☐ `this union that`

☐ `left union right`

☐ `(left union right) incl that`

☐ `((left union right) union that) incl elem`

(g) Assuming `remove(n: Int)` should return an `IntSet` that does not contain `n`, but contains all other elements of `this`, what goes in (7), (8)?

☐ `left, right` ☐ `left.incl(n), right` ☐ `left, right.incl(n)`

☐ `left.remove(n), right` ☐ `left, right.remove(n)`

(h) What goes in (9), (10)?

☐ `left, right` ☐ `left.incl(n), right` ☐ `left, right.incl(n)`

☐ `left.remove(n), right` ☐ `left, right.remove(n)`

(i) What goes in (11)?

☐ `this`

☐ `this union that`

☐ `left union right`

☐ `(left union right) incl that`

☐ `((left union right) union that) incl elem`

13. (9 pts) **Latency and fault-tolerance**.

(a) *Latency* is degradation in performance due to... (select two)

- ☐ a small number of cores in the central processing unit
- ☐ slow data transfer across the network or cluster
- ☐ shuffling data between different nodes in a cluster
- ☐ stack overflow caused by recursion

(b) Hadoop achieves fault-tolerance by...

- ☐ using lazy evaluation and garbage collection.
- ☐ writing intermediate computations to disk.
- ☐ keeping all data immutable and in-memory.
- ☐ replaying functional transformations over the original (immutable) dataset.

(c) Which is **not** one of the ways Spark decreases latency while remaining fault-tolerant?

- ☐ using ideas from functional programming.
- ☐ using ideas from imperative programming; e.g., mutation and side effects.
- ☐ keeping all data immutable and in-memory.
- ☐ replaying functional transformations over the original (immutable) dataset.

14. (12 pts) **Transformations and actions**.

(a) A **transformation** on an RDD... (select two)

- ☐ does not immediately compute a result.
- ☐ immediately computes and returns a result.
- ☐ is lazily evaluated.
- ☐ is eagerly evaluated.

(b) An **action** on an RDD... (select two)

- ☐ does not immediately compute a result.
- ☐ immediately computes and returns a result.
- ☐ is lazily evaluated.
- ☐ is eagerly evaluated.

(c) After performing a series of transformations on an `RDD`, which of the following methods would ensure that Spark actually carries out the transformations.

☐ `mapValues()` ☐ `collect()` ☐ `groupBy()` ☐ none of these

(d) After performing a series of transformations on an `RDD`, which of the following methods could you use to make sure those transformations are not repeated unnecessarily?

☐ `save()` ☐ `persist()` ☐ `collect()` ☐ `parallelize()`

(e) Why does the `RDD` class have no `foldLeft` method?

- ☐ `foldLeft` is not stack-safe.
- ☐ `foldLeft` is not fault-tolerant.
- ☐ `foldLeft` only works on `PairRDD`s.
- ☐ `foldLeft` is not parallelizable.
- ☐ It's not true; the RDD class *does* have a `foldLeft` method.

(f) Which method of the `RDD` class has the same effect as `foldLeft` and overcomes limitations of the latter?

☐ `aggregate` ☐ `foldRight` ☐ `join` ☐ `leftOuterJoin` ☐ `collect`

– scratch –