Quiz #3

Started: Mar 31 at 9:58pm

Quiz Instructions

The questions are from Chapter 5 Strictness and laziness, Chapter 6 Purely Functional State, and Chapter 7 Purely functional parallelism.

Two attempts. The highest score counts.

Make sure you prepare for the quiz before taking the test. Pay very close attention to the concepts from the lecture notes.

Some questions have "multiple answers" (as opposed to the multiple-choice type) and a wrong answer will be penalized with a negative score.

Question 1 — 10 pts

Consider method foldRight in the Stream trait:

```
def foldRight[B](z: => B)(f: (A, => B) => B): B =
    this match {
      case Cons(h,t) => f(h(), t().foldRight(z)(f))
      case _ => z
    }
```

Under what conditions does foldRight stop processing the remainder of a stream and finishes early ?

- ○ The function stops if argument z is forced evaluation.
- ○ The function stops if f returns None.
- ○ The function stops if f gets its second argument by value.
- ◉ Recursion stops if f does not evaluate its second argument.
- ○ Recursion stops if f does not evaluate its first argument.

Question 2 — 10 pts

Consider this code that uses the *Stream* trait from the textbook:

```
val s = Stream(10,11,12,13).append(Stream(1, 2, 3))
val sum = s.foldRight(0)(_ + _)
```

Enter the total number of *Cons* nodes created by the execution of the two assignments. It is not 7.

4

Question 3

10 pts

Which of the following statements are true ?

- [ ] The correct syntax to declare a non-strict parameter p of type A is: p: =>A.
- [x] Executing assignment

  ```
  val st1 = cons(1, cons(2, cons(3, cons(4, empty))))
  ```

  allocates memory for 4 Cons nodes. (Use the textbook version of the Stream types).
- [x] The Stream (textbook version) *Cons* case class uses thunks for the head and the tail to delay their evaluation until they are referenced (i.e. needed).
- [x] The delayed evaluation of a lazy variable that uses only pure functions/methods called with referential transparent arguments could lead to side effects.
- [x] A lazy variable is evaluated only the first time it is referenced.

Question 4

10 pts

Consider the State[S,A] state action case class from the textbook and the Rand type alias defined as

```
type Rand[A] = State[RNG, A]
```

What is the correct expression missing from the definition of state action *rndLstElem* so that it generates a random element from a non-empty list *lst* when used with a RNG?

```
def rndLstElem[A](lst: List[A]): Rand[A] =
  ____________________________________________
```

- ○ nonNegativeInt(lst).map(index => lst(index))
- ○ double(lst).map(index => lst.get(index.toInt))
- ◉ nonNegativeLessThan(lst.size).map(lst(_))
- ○ nonNegativeInt(lst.size).flatMap(index => lst(index))

Question 5

10 pts

The modify method of the *State*[S,A] type (Chapter 6) looks like this:

```
def modify[S](f: S => S): State[S, Unit] = for {
  s <- get
  _ <- set(f(s))
} yield ()
```

It looks like it actually changes the state of an object, contrary to the fundamental rule of functional programming. Which statement below explains best what is really going on ?

- ○ Indeed, modify just changes the state, having obvious side effects.
- ○ modify has no side effects because it needs a state object, like RNG, to operate on.
- ○ modify has no side effect, since it completely replaces the entire state of the object, not just some fields.
- ◉ *modify* returns a state action that can be executed with this syntax, assuming a state object *s* and a transformation function *f*: S => S:

```
modify(f).run(s)
```

The *modify* function is desugared (i.e. compiled) to this code:

```
def modify[S](f: S => S): State[S, Unit] =
 get.flatMap(s => set(f(s)).map(p => ()))
```

There is no side effect going on in this function. The state is passed as parameter, and eventually passed to the caller.

Question 6

10 pts

For this question we use the State[S,A] class for representing state actions and the Rand[A] = State[RNG, A] type alias.

Which of the following state actions generate a *Double* number x, where $x \in [k, k + 0.5)$ and k is a random integer in the set {0, 1, 2, 3, 4, 5} ?

- [x]
```
val ra2 = for {
  d <- double
  i <- nonNegativeLessThan(6)
  x = d / 2 + i
} yield x
```

- [x]
```
val ra3 = double.map2(nonNegativeLessThan(6))((d, i) => d / 2 + i)
```

- [ ]
```
double.map(d => nonNegativeLessThan(6).map(i => i + d / 2))
```

- [x]
```
nonNegativeLessThan(6).flatMap(i =>
     double.map(d => i.toDouble + d / 2))
```

Question 7

10 pts

Consider the following code that uses the first version of the Par[A] type from Chapter 7:

```
val p: Par[Int] = lazyUnit(42)
val f: (Int => Par[String]) = n => lazyUnit(n.toString)
val s = flatMap(p)(f)
```

Then, computation ***p*** and the computation returned by function f are independent and will be executed **concurrently**.

- (●) True
- ( ) False

Question 8

10 pts

Suppose we have long running expressions *a: => A* and *b: => B*, and a function *f: (A, B) => C*. Which of the following is the correct implementation of a function *combine* that evaluates expressions *a* and *b* **concurrently, in parallel logical threads**, and then combines the results using function *f*?

- (●)
```
def combine[A,B,C](a: => A, b: => B)(f: (A, B) => C): Par[C] =
  map2(lazyUnit(a), lazyUnit(b))(f)
```
- ( )
```
def combine[A,B,C](a: Par[A], b: Par[B])(f: (A, B) => C): Par[C] =
  map2(unit(a), unit(b))(f)
```
- ( )
```
def combine[A,B,C](a: => A, b: => B)(f: (A, B) => C): Par[C] = {
  val pa = fork(unit(a))
  val pb = fork(unit(b))
  flatMap(pa)(a => map(b => f(a, b)))
```
- ( )
```
def combine[A,B,C](a: => A, b: => B)(f: (A, B) => C): Par[C] =
  map2(unit(a), unit(b))(f)
```
- ( )
```
def combine[A,B,C](a: => A, b: => B)(f: (A, B) => C): Par[C] =
  map(lazyUnit(a))(a => map(lazyUnit(b))(b => f(a, b)))
```

Question 9

10 pts

Consider this program that must evaluate in a variable *y* expression $\sqrt{3} + \sqrt{5}$ in parallel, by spawning a logical thread to compute each of the square roots:

```
val es = Executors.newFixedThreadPool(10)
val asyncSqrt: Double => Par[Double] = asyncF((x: Double) => math.sqrt(x))
val p3 = asyncSqrt(3.0)
val p5 = asyncSqrt(5.0)

val y = _____________________________  // missing code
```

Which of the following is the correct line that is missing ?

- ○ `p3.run + p5.run`
- ◉ `run(es)(p3).get + run(es)(p5).get`
- ○ `map2(p3, p5)(_ + _)`
- ○ `p3.get + p5.get`
- ○ `run(es)(p3) + run(es)(p5)`

Question 10

10 pts

Which of the following *Par* operations (using the first version from the textbook) extracts a value from a parallel computation by forcing its execution?

- ○ lazyUnit
- ○ unit
- ◉ run

- ○ map2
- ○ fork

Quiz saved at 10:56pm

Submit Quiz