Par: Library Design for Parallel and Asynchronous Computation

Read Chapter 7 before or during solving exercises. The chapter discusses two APIs that are very similar, but not identical. The simpler one is based on Java's `Future`. The problem with the API is that it cannot guarantee deadlock-freeness when the number of threads in the pool is limited. The non-blocking API is introduced in Section 7.3.4, and all the exercise below assume the use of the non-blocking API (with continuations). I chose to work with the non-blocking API, even if it is slightly more complex, to avoid deadlocks in tests (which would cause a lot of frustration, whenever you make a mistake).

If the test suite behaves weirdly (you are certain that the solutions are good but the tests are failing), try increasing the `TIMEOUT` value in the test file. Your computer may be too slow, or too fast, to work with the suggested configuration. If the tests deadlock, it is likely that the solution is broken.

We continue to use standard library lists (not fpinscala/adpro): https://www.scala-lang.org/api/current/scala/collection/immutable/List$.html

Hand-in `Exercises.scala`. Exercise 5 is the most important so do not skip it.

Exercise 1. The book introduces the following function on the basic type `Par` in Section 7.1.1 (we use the variant from Section 7.3.4, but this question is about the variant from Section 7.1.1):

```
def unit[A](a: =>A): Par[A]
```

Explain in English why the argument `a` is passed by-name to the `unit` function. Write a short explanation (max 5 lines) in the dedicated comment in `Exercises.scala`.

Exercise 2. Use `lazyUnit` to write a function that converts any function `A => B` to one that evaluates its result asynchronously (it spawns a separate thread).[1]

```
def asyncF[A,B](f: A =>B): A =>Par[B]
```

Exercise 3. Find the implementation of `map` in the chapter (Section 7.2.1). Write in English how would you test it. What is a test-case and how would you decide that the test-case has passed. Write your response in English in the designated comment in the Scala file.

Expected size: 10-15 lines of text, but more (and less) is allowed.

Exercise 4. Write a function `sequence` that takes a list of parallel computations (`List[Par[B]]`) and returns a parallel computation producing a list (`Par[List[B]]`).[2]

```
def sequence[A](ps: List[Par[A]]): Par[List[A]]
```

Note: `sequence` should not execute (force) any of the computations on the argument list. It should just 'repackage' the entire thing as a single parallel computation producing a list. Why is this useful? This allows us to continue building a computation on the entire list (for instance using `map` or `sortPar` from the chapter) without waiting for all the elements to terminate.

Exercise 5. Write a short function `wget` that, given a (variable length argument) list of URIs, downloads the files at URIs and returns the strings representing the HTML files at these addresses. For instance, we could call:

[1] Exercise 7.4 [Pilquist, Chiusano, Bjarnason, 2022]
[2] Exercise 7.5 [Pilquist, Chiusano, Bjarnason, 2022]

```
wget("https://www.dr.dk", "https://www.itu.dk")
```

This should return a list of two strings, the first one containing the file returned by DR, the second by ITU. You can use `scala.io.Source.fromURL` to download the file.[3] It is important that downloading proceeds in parallel (so all the files in a list of arguments are downloaded over parallel connections), using the `Par` API.

There are no real tests for this exercise, but the test `"Exercise 5"` simply attempts to download three websites using your implementation of `wget` and prints the first line (up to 100 chars) in the test log. It will fail, if your code throws an exception.

Experiment with downloading sequentially and in parallel by modifying the test code. Can you observe a difference in speed?

If tests hand (your function hangs after download), make sure to call `shutdown` on the executor service before returning.

Important: In a comment below the implementation of `wget` explain in a few sentences how your implementation achieves concurrency.

Exercise 6. Implement `parFilter`, which filters elements of a list in parallel (so the predicate `f` is executed in parallel for the various lists elements). [4]

```
def parFilter[A](as: List[A])(f: A =>Boolean): Par[List[A]]
```

Exercise 7. Section 7.4 discusses the implementation of `choice`, a combinator that first computes a Boolean condition and returns result of one parallel computation if the condition is true, and of another computation if the condition is false. This operator can be used to spawn a computation based on a result of a Boolean condition. However the Boolean operation only allows to choose between two forks, but it might be useful to make a choice between `N` operators.

Implement `choiceN` and then `choice` in terms of `choiceN`.[5]

```
def choiceN[A](n: Par[Int])(choices: List[Par[A]]): Par[A]
def choice[A](cond: Par[Boolean])(t: Par[A], f: Par[A]): Par[A]
```

Exercise 8. Implement a general parallel computation `chooser`, and then use it to implement `choice` and `choiceN`. A `chooser` uses a parallel computation to obtain a selector for one of the available parallel computations in the range provided by `choices`. We implement this as an extension method on `Par[A]` with the following type:[6]

```
def chooser[B](choices: A =>Par[B]): Par[B]
```

Note: In search for an "a-ha-moment," compare its type with the types of `Option.flatMap`, `Stream.flatMap`, `List.flatMap` and `State.flatMap`. Observe that the `chooser` is used to compose (sequence) to parallel computations here.

Exercise 9. Implement `join` using `chooser`. Compare the type of `join` with the type of `concat` in week 2.[7]

[3] We have reports, at least on Windows and Macs, that `fromURL` hangs or crashes if you omit the codepage parameter, so add it like here: https://stackoverflow.com/questions/29987146/using-result-from-scalas-fromurl-throws-exception.

[4] Exercise 7.6 [Pilquist, Chiusano, Bjarnason, 2022]

[5] Exercise 7.11 [Pilquist, Chiusano, Bjarnason, 2022]

[6] Exercise 7.13 [Pilquist, Chiusano, Bjarnason, 2022]

[7] Exercise 7.14 [Pilquist, Chiusano, Bjarnason, 2022]

```
def join[A](a: Par[Par[A]]): Par[A]
```

After having done that assume for a moment that `join` is a primitive, as it could be. Implement `chooser` using `join` and `map`. To keep the exercise solution local, let's make this new version of `chooser` a function, not a method. Do not invoke the other `chooser` directly.

```
def chooser[A, B](pa: Par[A])(f: A =>Par[B]): Par[B]
```

Exercise 10. This exercise has nothing to do with parallelism, but it trains the general skill of expanding types using extension methods. Implement an extension method for `Par[Par[A]]` in Scala so that we can call `join` as a method (delegate to the function `join`).

```
def join: Par[A]
```

Similarly, implement extension methods for `choice` and `choiceN` by delegating to their function variants from previous exercises. It is instructive to check the types of the functions in Exercise 7 above.

```
def choiceN[A](choices: List[Par[A]]): Par[A]
def choice[A](t: Par[A], f: Par[A]): Par[A]
```

Explain briefly in English why all three of these extensions cannot be placed in the same extension block? Why they cannot be put in the same block with extensions for `Par[A]`?