built on 2023/10/25 at 16:01:31 ***due:*** *mon nov 06 @ 11:59pm*

This assignment aims to give you more experience writing code that takes advantage of concurrency. We're providing some starter files; please *download the starter code from the course website.*

You will zip all your work in one zip file and upload it to Canvas before the due date.

IMPORTANT: Your implementation *no longer* has to strictly follow the functional style. But in places that make sense, the functional style is still encouraged (as it can greatly reduce unintended programming errors).

- You can define as many helper functions as necessary. Be mindful of what you should expose to outside your function.
- You are going to be graded on style as well as on correctness.
- Test your code!

Task 1: Promise Me Future Data Crunching (8 points)

For this task, save your code in `crunch/src/main/scala/DataCrunch.scala`

You will implement a few functions related to data crunching using future and promise. All your implementations for this problem will go inside `object DataCrunch`. The object does *not* extend `App`, but a `main` function is provided.

The trait `DataProvider`, reproduced below from the stub file, describes the kind of interface for retrieving data we're working with in this problem:

```
trait DataProvider {
  def get(onSuccess: Seq[String] => Unit,
          onFailure: () => Unit): Unit
}
```

To motivate how this interface will be used, consider the task of reading from a text file and delivering the lines to the consumer of the file's data. The following interaction is natural for a callback-style program:

```
def funcOnSuccess(lines: Seq[String]) = lines.foreach(println(_)) // 1
def funcOnFailure() = println("Failed")                           // 2
val fileProvider: DataProvider = FileSource("/bigdata/lorem.txt") // 3
fileProvider.get(funcOnSuccess, funcOnFailure)                    // 4
```

The code on lines 1–2 define two functions to be called when the data provider successfully obtain the lines and when the data provide fails. Specifically, `funcOnSuccess` prints out all the lines being passed in. The code on line 3 creates a data provider. Line 4 is where we use the interface to register what to do on success and in the event of a failure, respectively.

For testing, the starter pack has a few objects implementing this `DataProvider` trait.

You're to write two functions (write helper functions as appropriate):

(1) Implement a function

```
def dataProviderFuture(dp: DataProvider): Future[Seq[String]]
```

that takes a data provider and *immediately* returns a `Future` representing the outcome of this data provider. That is, if the data provider succeeds, the future will, after the data provider finishes,

contain the lines provided—and if the data provider fails, the future will hold the exception `Exception("failed")`.

NOTE: You must *not* use `Await` and must *not* spin wait.

(2) Implement a function

```
def highestFreq(linesFut: Future[Seq[String]]): Future[(String, Double)]
```

that takes a `Future` of a string sequence and returns a `Future` of a string/double pair. This pair (w, f) encodes two pieces of information: w is the word that has the highest frequency in the given lines. If there are multiple words with the same frequency, return any of them. In addition to this, f is the the popularity fraction given by

$$f = \frac{\text{the count of this word}}{\text{total word count}}$$

To simplify matters, use the following code to separate a string into words: If `ls` is a string representing one line, the following expression evaluates to an "array" of nonempty words on this line:

```
ls.split("\\s+").filter(_.nonEmpty)
```

Example: Running `highestFreq` on the provided `LoremIpsum` object should yield `(a,0.02912...)`—because `a` is the most frequent word, appearing 3 times.

NOTE: For this part, you must *not* use `Await` and must *not* spin wait.

Task 2: Top-k Words (4 points)

For this task, save your code in `topk/src/main/scala/TopK.scala`

You are to implement a function

```
def topKWords(k: Int)(fileSpec: String): Vector[(String, Int)]
```

that takes a filename (e.g., `../test.txt`) and returns a vector of `(word, freq)` of the top-`k` most-frequent words. The goal of this function is to:

- Read the given file and determine the top-`k` most-frequent words. If there are multiple words with the same frequency, break ties lexicographically (i.e., order words of the same frequency by dictionary order).
- Make use of parallelism/concurrency. While file reading is sequential in nature, the lines can be split and tokenized in parallel.

You should be using an execution pool or `Future`s. Lines in a file could be of varying lengths. Think carefully how the code should split up work to maximize parallelism benefits.

In this task, we define words as follows: the following code to separate a string into words: If `ls` is a string representing one line, the following expression evaluates to an "array" of nonempty tokens on this line:

```
ls.split("\\s+").filter(_.nonEmpty)
```

We will further downcase (convert it to lowercase letters) every token. For this task, a word is a token that (i) begins with `a-z`, (ii) contains no digits (`0-9`), (iii) and contains only letter in the English alphabet (`a-z`) although it may have up to two dash (`-`) inside it (e.g., `user-generated`).

Task 3: Concurrent Web Crawler (12 points)

For this task, save your code in `crawler/src/main/scala/Crawler.scala`

Some of you have written a web crawler in Java using a combination of the Jsoup and Apache Commons Http packages. In this task, you're going to do the same in Scala, with one additional feature: we'll try to crawl many pages and parse them simultaneously in parallel.

More specifically, you will write a function

```
def crawlForStats(basePath: String): WebStats
```

which takes a `basePath` string representing the starting point and base path for crawling (more on this later) and returns a case-class `WebStats` that stores info about what we've crawled:

```
sealed case class WebStats(
  // the total number of (unique) files found
  numFiles: Int,
  // the total number of (unique) file extensions (.jpg is different from .jpeg)
  numExts: Int,
  // a map storing the total number of files for each extension.
  extCounts: Map[String, Int],
  // the total number of words in all html files combined, excluding
  // all html tags, attributes and html comments.
  totalWordCount: Long
)
```

For word count, your code will downcase the string (so `Ant` and `ant` are the same word) and only strings beginning with `a-z` will be counted. Other than that, our definition of what constitutes a word is pretty loose—do what is reasonable. For unique extensions, we downcase all extensions, so `jpeg` and `JPEG` are the same.

As a familiar example, using `basePath = "https://cs.muic.mahidol.ac.th/courses/ooc/api/"` means starting the crawl at this location and only retrieving pages that begin with this base path. It is important that the code doesn't crawl any links that don't begin with this base path; otherwise, we might be crawling a large portion of the Internet.

When we a crawl a page p, we parse the page entirely and recursively crawl all (hyper)links that p points to as long as they are under the given base path.

Concurrency. To avoid crawling the same page multiple times, one effective strategy is to apply breadth-first search (BFS), so only unique pages are crawled. But then where is an opportunity for parallelism/concurrency? Given a frontier, the next frontier can be derived in parallel. In particular, each of the pages in the frontier can be crawled simultaneously (taking advantage of a execution pool or `Future`'s).

To implement this, we have a couple of choices in terms how the next frontier will be derived and stored. The next frontier could be a concurrent data structure, allowing multiple threads to try to add to it at the same time. Differently, the next frontier could be derived from a collection of `Future`'s, each keeping the result from crawling a page in the frontier.

Tips. Jsoup is a friendly library. To allow you to focus mainly on the concurrency aspect, we have written some demo functions (`Demo.scala`) that use Jsoup to print out all the links—and the textual portion of that page (for use in word count).

Furthermore, pay attention to the following points:

- *Links that are relative to the current page.* For example, `https://ex.com/hello/test.html` can have a link `../img/b.jpg`, which refers to `https://ex.com/img/b.jpg`. Following relative links will be necessary for your code to work. Luckily, the built-in `java.net.URL` class knows how to do this already, so you just have use it.
- *Links that contain fragments.* For example, `https://ex.org/a.html#blah` and `https://ex.org/a.html?o=blah`). When we talk about unique pages, we ignore these fragments. Hence, the two sample links are the same page (`https://ex.org/a.html`). Once again, use the `URL` class to handle this.