CSE250, Spring 2022 Assignment 1 Due Wed. Feb. 16, 11:59pm

Lectures and Reading:

Chapters 2–4 of the textbook are less "fact-dense" than Chapter 1. More space is devoted to longer examples. Not all examples can be covered in lectures, and those that are may be streamlined at least on slides. Notes posted in advance may give text (sub-)sections to focus on. For Friday, those sections are the discussion and boxes centered on pages 56–57. For next week in chapter 3, note subsections 3.1.1, 3.2.1, the box on page 87, and 3.2.1.1 as a "word-to-the-wise," but the main coverage will be subsections 3.2.3, 3.2.4, and 3.2.5. Then section 3.3 about the companion object, some of which has already been touched on. Section 3.5 will be at less detail than the text—I'll avoid going into "`unapply`" (and wth are "`curried`" and "`tupled`" anyway). Chapter 4 will be covered pretty much whole, but it is tricky to lecture on because whether you have had Java makes a big difference to the speed of uptake. I will try a one-slide-per-thing pace (into week 4), maybe a little fast but accessible as a first look while not boring those who have more experience with it. The long section 4.2—before it gets to *parametric* (i.e., generic or templated) polymorphism—ought to be familiar from previous experience of OOP in any language. (Note it talks about the square-versus-rectangle issue.)

——— *Assignment 1, due Feb, 16 "midnight stretchy" on CSE Autograder* ———

Brief Task Statement:

Read a text file and split each line into items separated by whitespace. Find the longest item, and also give what line it is on, and its index among items in that line. Make sure that the empty string is not counted as an item, nor any whitespace-only string.

For example, if the file is the code `AHquiz.scala` given for the week-2 recitation exercise, then the longest item is the chained method call `List.tabulate(10)(_+1).zip(answers)` toward the end of the file. It is the fourth item on its line, so its zero-based index is 3. However, because the line is indented, you might get an unwanted initial blank item if you don't filter those out, which would make the index report as 4 (or higher).

File(s) to Submit (now a zipfile is expected):

- `MaxWords.scala`
- Optionally, an example of output `output.txt` from a run of your program, on `AHquiz.scala` or your own source code or an optional input file of your choosing.
- Optionally, a file `essay.txt` with your short essay answer, or that can go as a comment at the end of your `MaxWords.scala`, or you can make your code write it to the end of `output.txt`. In any way, it should begin with the word `Essay`.

Note that we will do "hidden tests" on our own "`words.txt`" files, passing them as command-line arguments if you do that. Fully-working code is worth **48 points** on the homework scale, plus **12 points** for the essay-type question below.

Specific Directions (note possible tweaks, which could follow after our Friday TA meeting):

- There is no pre-given code for this assignment. (When we hit ADTs, there will be.)
- Testing add-ons (as used in CSE116) not required, but "primitive" print-debugging of code at partial stages highly recommended—to observe the types you get.
- **Tweak**: Command-line arguments are already enabled when you extend `App`. So make the first two lines of your object (after a header comment and `import` statements) be:

  ```
  object MaxWords extends App {
     val inputFile = if (args.length >= 1) args(0) else "words.txt"
  ```

  Then use `inputFile` as the argument to `Source.fromFile` imitating the text on p35.
- *Thou shalt not* use screen input for the filename, or for anything in a submitted file.
- **Tweak**: submitting a zipfile is expected.
- Your answer should print to a file `output.txt` in your project root folder when your program is run. It should have the form typified by

  ```
  The word of longest length 35 is List.tabulate(10)(_+1).zip(answers) in line 91, column 3
  ```

  in the above example. In case of ties, you may choose the first or last or any longest word—and should say in (a comment and in) your essay which you give.
- As this exemplifies, it is AOK to keep punctuation marks that touch your words and count them in the length.
- Your essay answer can be in a `/* .. */` comment at the bottom of your main source file, *or* you may have your program itself print it to `output.txt`. (This does not require you to submit `output.txt`; we will generate it by running your code. This may be tweaked to allow an option of having in a separate file that is zipped, in which case it would be AOK to include an example fo your `output.txt`.)
- To read the file, it is recommended to imitate the syntax on the bottom of page 35, except: split on whitespace not comma by doing `split("\\s+")`, don't include the mapping `toDouble`, and here-or-somewhere, *do* `filter` with a function that removes empty strings. (Shortcutting the text's use of "rocket" notation via doing `_.split` is OK here, strictly optional.)

Essay Question: Was the Scala `List` type involved at any stage of your processing, possibly under the hood? Incorporate this into a paragraph describing your algorithmic strategy in general, including how (or whether) you dealt with *tuples*, which are different from lists.

Curveballs to Watch:

One convenience by which Scala resembles Python is being able to return multiple values in a tuple—even when the values mix types like having integers and strings. If you declare `var x` and `var y`, however, you can't do the "Pythonic multi-assignment" `(x,y) = (a,b)`, say (which can be done even without the parens). In Scala, you *can* repeat `var` declarations, so `var (x,y) = (a,b)` is OK even multiple times. But the more "Scalastic" way IMPHO is to name the tuple. E.g.:

```
var x = 2
var y = 3
var t = (x,y)
...
t = (4,5)    //OK
```

One further thing to note, though, is that this did not assign 4 to `x` and 5 to `y`. It read values from them to build `t` originally as $(2, 3)$, but then the next assignment only changes `t`. It seems not to copy references even if `x` and `y` are arrays created with `new`. You can get the individual values by `t._1` and `t._2`, and so on for longer tuples. (Note they have 1-based not 0-based indexing.) [Or you can **extract** them by a `match` expression using `t` on the right-hand side, but then the left-hand side is implictly a `val`. That ultimately explains why Scala breaks on "`(x,y) = (a,b)`"—it thinks you are trying to do a `match` but wrongly with `var`.]

- I've seen that a recent CSE116 had an assignment in which a file was read into a single string including `\n` for carriage returns. If you started that way, you would have to hunt for them—and have to make sure that `.split` on whitespace didn't vanish them entirely. The indicated way from our textbook gives you a matrix of type `Array[Array[String]]`. Dealing with this doubledecker structure is the main challenge of the assignment, but that structure is really there to help not hinder.
- Likewise, this project does not intend you to "flatten" the file down to a single array of strings, nor a single list of strings.
- The use of chained methods—some as "higher-order functions"—can get funky but the task tries to stay close to text examples.

Learning Objectives:

- Experience with procedural code in Scala: loops and if-then-else.
- Experience with reading (potentially large) text files in Scala.
- Experience with methods covered in sections 1.7 and 1.8 of the text, including some element-wise functions.
- Experience with tuples—as opposed to lists.
- Experience with data having multiple levels—here, array-of-array.
- Writing a short app from scratch.

On the Horizon:

This assignment may be progressed into one that really works on words. One idea is to measure how often a writer puts a long word next to short words—when a less ginormously brobdingnagian synonym might flow better. Metrics for that would like to flow over a single whole list of text. They would take two or three words at a time—thus leading to another good question in class today (2/9) about doing recursion on lists with bigger base-cases than `Nil` and (hence) recursion taking multiple elements at a time.

But before even thinking of that, there's the issue of how you would re-locate the long words you want to edit. If you've flattened everything into one list, you've lost the line-number-and-place structure that the original text file had. You could re-print it with a given word-wrap threshold, but that doesn't help you re-locate the long words without re-doing lots of sequential searching. So you'd want to retain the original array structure and copy the words into lists for processing the metrics. Then how do you remember which indices go with which words? One way is to pull off a list of tuples rather than just a list of words. But then you'd have to modify the element-wise string functions to take tuples and fetch the string part of the tuples.

At any rate, that would promote the app into one where arrays, lists, and tuples all work in concert. That also, however, gets toward the limit of one-shot jimmy-rigging of code. We'd want to make an ADT that could call those kinds of shots in a more-encapsulated manner. That is: put the tuple-shuffling under the hood, until you get just the tuples showing the long words causing the highest metric scores and where they are in the matrix of strings.

- And for a lower-level matter, it would be nice to make punctuation marks split off by themselves. That gets into the realm of building a **tokenizer**. This is about the easiest task in a compilers course but a little much for a first assignment—so I hoped to find one built-in for Scala. The closest is that Scala could call `java.util.StringTokenizer`, described here, whose version with the flag set true gives exactly the behavior we'd want. But that page says, "`StringTokenizer` is a legacy class that is retained for compatibility reasons although its use is discouraged in new code." It further recommends using (Java's version of) `split`, but I don't see how to make it work with one-liner convenience. The roll-your-own tokenizer I've thought of is another example of list recursion taking more than one item at a time, and with the "accumulator" recursion pattern to boot—so tabled for later when we cover recursion officially.

- Similarity metrics on adjacent words in a sequence lead into some serious application topics, such as in genomics. My CSE250 in 2014 had an assignment involving Hamming distance and some broader notions of edit distance. But if we wanted to go in a less-serious direction, dare I say...

- Wordle?