Assignment 6 (Scala)

"The most effective debugging tool is still careful thought, coupled with judiciously placed print statements."

— Brian W. Kernighan, in Unix for Beginners (1979)

This assignemnt is about Scala and worth 10%. The first and second part are due on 16 November at 11pm, and the third part on 21 December at 11pm. You are asked to implement two programs about list processing and recursion. The third part is more advanced and might include material you have not yet seen in the first lecture.

Important:

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the commandline.[1] Use the template files provided and do not make any changes to arguments of functions or to any types. You are free to implement any auxiliary function you might need.
- Do not leave any test cases running in your code because this might slow down your program! Comment test cases out before submission, otherwise you might hit a time-out.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Array`s or `ListBuffer`s, for example.
- Do not use `return` in your code! It has a different meaning in Scala than in Java.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each part will be restricted to a maximum of 30 seconds on my laptop.

Disclaimer

It should be understood that the work you submit represents your **own** effort! You have not copied from anyone else. An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

[1] All major OSes, including Windows, have a commandline. So there is no good reason to not download Scala, install it and run it on your own computer. Just do it!

Reference Implementation

Like the C++ assignments, the Scala assignments will work like this: you push your files to GitHub and receive (after sometimes a long delay) some automated feedback. In the end we take a snapshot of the submitted files and apply an automated marking script to them.

In addition, the Scala assignments come with a reference implementation in form of a `jar`-file. This allows you to run any test cases on your own computer. For example you can call Scala on the command line with the option `-cp collatz.jar` and then query any function from the template file. Say you want to find out what the functions `collatz` and `collatz_max` produce: for this you just need to prefix them with the object name `CW6a` (and `CW6b` respectively for `drumb.jar`). If you want to find out what these functions produce for the argument `6`, you would type something like:

```
$ scala -cp collatz.jar

scala> CW6a.collatz(6)
...
scala> CW6a.collatz_max(6)
...
```

Part 1 (3 Marks, file collatz.scala)

This part is about recursion. You are asked to implement a Scala program that tests examples of the $3n + 1$-*conjecture*, also called *Collatz conjecture*. This conjecture can be described as follows: Start with any positive number n greater than 0:

- If n is even, divide it by 2 to obtain $n/2$.
- If n is odd, multiply it by 3 and add 1 to obtain $3n + 1$.
- Repeat this process and you will always end up with 1.

For example if you start with 6, or 9, you obtain the series

$$6, 3, 10, 5, 16, 8, 4, 2, 1 \quad (= 8 \text{ steps})$$
$$9, 28, 14, 7, 22, 11, 34, 17, 52, 26, 13, 40, 20, 10, 5, 16, 8, 4, 2, 1 \quad (= 19 \text{ steps})$$

As you can see, the numbers go up and down like a roller-coaster, but curiously they seem to always terminate in 1. The conjecture is that this will *always* happen for every number greater than 0.[2]

Tasks

[2]While it is relatively easy to test this conjecture with particular numbers, it is an interesting open problem to *prove* that the conjecture is true for *all* numbers (> 0). Paul Erdös, a famous mathematician you might have hard about, said about this conjecture: "Mathematics may not be ready for such problems." and also offered a $500 cash prize for its solution. Jeffrey Lagarias,

(1) You are asked to implement a recursive function that calculates the number of steps needed until a series ends with 1. In case of starting with 6, it takes 8 steps and in case of starting with 9, it takes 19 (see above). In order to try out this function with large numbers, you should use `Long` as argument type, instead of `Int`. You can assume this function will be called with numbers between 1 and 1 Million. [2 Marks]

(2) Write a second function that takes an upper bound as argument and calculates the steps for all numbers in the range from 1 up to this bound. It returns the maximum number of steps and the corresponding number that needs that many steps. More precisely it returns a pair where the first component is the number of steps and the second is the corresponding number. [1 Mark]

Test Data: Some test ranges are:

- 1 to 10 where 9 takes 19 steps
- 1 to 100 where 97 takes 118 steps,
- 1 to 1,000 where 871 takes 178 steps,
- 1 to 10,000 where 6,171 takes 261 steps,
- 1 to 100,000 where 77,031 takes 350 steps,
- 1 to 1 Million where 837,799 takes 524 steps

Hints: useful math operators: `%` for modulo; useful functions: `(1 to 10)` for ranges, `.toInt`, `.toList` for conversions, `List(...).max` for the maximum of a list, `List(...).indexOf(...)` for the first index of a value in a list.

Part 2 (3 Marks, file drumb.scala)

A purely fictional character named Mr T. Drumb inherited in 1978 approximately 200 Million Dollar from his father. Mr Drumb prides himself to be a brilliant business man because nowadays it is estimated he is 3 Billion Dollar worth (one is not sure, of course, because Mr Drumb refuses to make his tax records public).

Since the question about Mr Drumb's business acumen remains open, let's do a quick back-of-the-envelope calculation in Scala whether his claim has any merit. Let's suppose we are given $100 in 1978 and we follow a really dumb investment strategy, namely:

---

another mathematician, claimed that based only on known information about this problem, "this is an extraordinarily difficult problem, completely out of reach of present day mathematics." There is also a xkcd cartoon about this conjecture (click here). If you are able to solve this conjecture, you will definitely get famous.

- We blindly choose a portfolio of stocks, say some Blue-Chip stocks or some Real Estate stocks.
- If some of the stocks in our portfolio are traded in January of a year, we invest our money in equal amounts in each of these stocks. For example if we have \$100 and there are four stocks that are traded in our portfolio, we buy \$25 worth of stocks from each. Be careful to also test cases where you trade with 3 stocks, for example.
- Next year in January, we look at how our stocks did, liquidate everything, and re-invest our (hopefully) increased money in again the stocks from our portfolio (there might be more stocks available, if companies from our portfolio got listed in that year, or less if some companies went bust or were de-listed).
- We do this for 40 years until January 2018 and check what would have become out of our \$100.

Until Yahoo was bought by Altaba this summer, historical stock market data for such back-of-the-envelope calculations was freely available online. Unfortuantely nowadays this kind of data is difficult to obtain, unless you are prepared to pay extortionate prices or be severely rate-limited. Therefore this coursework comes with a number of files containing CSV-lists with the historical stock prices for the companies in our portfolios. Use these files for the following tasks.

Tasks

(1) Write a function `get_january_data` that takes a stock symbol and a year as arguments. The function reads the corresponding CSV-file and returns the list of strings that start with the given year (each line in the CSV-list is of the form `someyear-01-someday,someprice`). [1 Mark]

(2) Write a function `get_first_price` that takes again a stock symbol and a year as arguments. It should return the first January price for the stock symbol in the given year. For this it uses the list of strings generated by `get_january_data`. A problem is that normally a stock exchange is not open on 1st of January, but depending on the day of the week on a later day (maybe 3rd or 4th). The easiest way to solve this problem is to obtain the whole January data for a stock symbol and then select the earliest, or first, entry in this list. The stock price of this entry should be converted into a double. Such a price might not exist, in case the company does not exist in the given year. For example, if you query for Google in January of 1980, then clearly Google did not exist yet. Therefore you are asked to return a trade price with type `Option[Double]`...`None` will be the value for when no price exists; `Some` if there is a price. [1 Mark]

(3) Write a function `get_prices` that takes a portfolio (a list of stock symbols), a years range and gets all the first trading prices for each year in the range.

You should organise this as a list of lists of `Option[Double]`'s. The inner lists are for all stock symbols from the portfolio and the outer list for the years. For example for Google and Apple in years 2010 (first line), 2011 (second line) and 2012 (third line) you obtain:

```
List(List(Some(311.349976), Some(20.544939)),
     List(Some(300.222351), Some(31.638695)),
     List(Some(330.555054), Some(39.478039)))
```

[1 Marks]

Advanced Part 3 (4 Marks, continue in file drumb.scala)

Tasks

(4) Write a function that calculates the *change factor* (delta) for how a stock price has changed from one year to the next. This is only well-defined, if the corresponding company has been traded in both years. In this case you can calculate

$$\frac{price_{new} - price_{old}}{price_{old}}$$

If the change factor is defined, you should return it as `Some(change_factor)`; if not, you should return `None`. [1 Mark]

(5) Write a function that calculates all change factors (deltas) for the prices we obtained under Part 2. For the running example of Google and Apple for the years 2010 to 2012 you should obtain 4 change factors:

```
List(List(Some(-0.03573992567129673), Some(0.539975124774038))
     List(Some(0.10103412653643493), Some(0.24777709700099845)))
```

That means Google did a bit badly in 2010, while Apple did very well. Both did OK in 2011. Make sure you handle the cases where a company is not listed in a year. In such cases the change factor should be `None` (see (4)). [1 Mark]

(6) Write a function that calculates the "yield", or balance, for one year for our portfolio. This function takes the change factors, the starting balance and the year as arguments. If no company from our portfolio existed in that year, the balance is unchanged. Otherwise we invest in each existing company an equal amount of our balance. Using the change factors computed under Task 2, calculate the new balance. Say we had $100 in 2010, we would have received in our running example involving Google and Apple:

```
$50 * -0.03573992567129673 + $50 * 0.539975124774038
                                        = $25.21175995513706
```

as profit for that year, and our new balance for 2011 is \$125 when converted to a `Long`. [1 Mark]

(7) Write a function that calculates the overall balance for a range of years where each year the yearly profit is compounded to the new balances and then re-invested into our portfolio. For this use the function and results generated under (6).

[1 Mark]

Test Data: File `drumb.scala` contains two portfolios collected from the S&P 500, one for blue-chip companies, including Facebook, Amazon and Baidu; and another for listed real-estate companies, whose names I have never heard of. Following the dumb investment strategy from 1978 until 2018 would have turned a starting balance of \$100 into roughly \$101,589 for real estate and a whopping \$1,587,528 for blue chips. Note when comparing these results with your own calculations: there might be some small rounding errors, which when compounded lead to moderately different values.

Hints: useful string functions: `.startsWith(...)` for checking whether a string has a given prefix, `_ ++ _` for concatenating two strings; useful option functions: `.flatten` flattens a list of options such that it filters way all `None`'s, `Try(...) getOrElse ...` runs some code that might raise an exception—if yes, then a default value can be given; useful list functions: `.head` for obtaining the first element in a non-empty list, `.length` for the length of a list.

Moral: Reflecting on our assumptions, we are over-estimating our yield in many ways: first, who can know in 1978 about what will turn out to be a blue chip company. Also, since the portfolios are chosen from the current S&P 500, they do not include the myriad of companies that went bust or were de-listed over the years. So where does this leave our fictional character Mr T. Drumb? Well, given his inheritance, a really dumb investment strategy would have done equally well, if not much better.