Core Part 3 (Scala, 3 Marks)

"[Google's MapReduce] abstraction is inspired by the map and reduce primitives present in Lisp and many other functional languages."

— Dean and Ghemawat, who designed this concept at Google

Important

- Make sure the files you submit can be processed by just calling `scala <<filename.scala>>` on the command line.[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 out test cases 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. It changes the meaning of your program, and you should never use it.
- 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 implemented the code entirely on your own. You have not copied from anyone else. Do not be tempted to ask Github Copilot for help or do any other shenanigans like this! 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 command line. So there is no good reason to not download Scala, install it and run it on your own computer. Just do it!

Reference Implementation

This Scala assignment comes with two reference implementations in form of `jar`-files. 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 postfix.jar` and then query any function from the `postfix.scala` file (similarly for file `postfix2.scala`). As usual you have to prefix the calls with `C3a` and `C3b`, respectively.

```
$ scala -cp postfix.jar

scala> C3a.syard(C3a.split("( 5 + 7 ) * 2"))
val res0: C3a.Toks = List(5, 7, +, 2, *)
```

Hints

For Core Part 3: useful operations for determining whether a string is a number are `.forall` and `.isDigit`. One way to calculate the the power operation is to use `.pow` on `BigInts`, like `BigInt(n).pow(m).toInt`.

Core Part (3 Marks, files postfix.scala, postfix2.scala)

The Shunting Yard Algorithm has been developed by Edsger Dijkstra, an influential computer scientist who developed many well-known algorithms. This algorithm transforms the usual infix notation of arithmetic expressions into the postfix notation, sometimes also called reverse Polish notation.

Why on Earth do people use the postfix notation? It is much more convenient to work with the usual infix notation for arithmetic expressions. Most modern pocket calculators (as opposed to the ones used 20 years ago) understand infix notation. So why on Earth? ...Well, many computers under the hood, even nowadays, use postfix notation extensively. For example if you give to the Java compiler the expression $1 + ((2 * 3) + (4 - 3))$, it will generate the Java Byte code

```
ldc 1
ldc 2
ldc 3
imul
ldc 4
ldc 3
isub
iadd
iadd
```

where the command `ldc` loads a constant onto the stack, and `imul`, `isub` and `iadd` are commands acting on the stack. Clearly this is the arithmetic expression in postfix notation.

The shunting yard algorithm processes an input token list using an operator stack and an output list. The input consists of numbers, operators ($+$, $-$, $*$, /) and parentheses, and for the purpose of the assignment we assume the input is always a well-formed expression in infix notation. The calculation in the shunting yard algorithm uses information about the precedences of the operators (given in the template file). The algorithm processes the input token list as follows:

- If there is a number as input token, then this token is transferred directly to the output list. Then the rest of the input is processed.
- If there is an operator as input token, then you need to check what is on top of the operator stack. If there are operators with a higher or equal precedence, these operators are first popped off from the stack and moved to the output list. Then the operator from the input is pushed onto the stack and the rest of the input is processed.
- If the input is a left-parenthesis, you push it on to the stack and continue processing the input.
- If the input is a right-parenthesis, then you pop off all operators from the stack to the output list until you reach the left-parenthesis. Then you discharge the (and) from the input and stack, and continue processing the input list.
- If the input is empty, then you move all remaining operators from the stack to the output list.

Tasks (file postfix.scala)

(1) Implement the shunting yard algorithm described above. The function, called `syard`, takes a list of tokens as first argument. The second and third arguments are the stack and output list represented as Scala lists. The most convenient way to implement this algorithm is to analyse what the input list, stack and output list look like in each step using pattern-matching. The algorithm transforms for example the input

```
List(3, +, 4, *, (, 2, -, 1, ))
```

into the postfix output

```
List(3, 4, 2, 1, -, *, +)
```

You can assume the input list is always a list representing a well-formed infix arithmetic expression. [1 Mark]

(2) Implement a compute function that takes a postfix expression as argument and evaluates it generating an integer as result. It uses a stack to evaluate the postfix expression. The operators $+$, $-$, $*$ are as usual; / is division on integers, for example $7/3 = 2$. [1 Mark]

Task (file postfix2.scala)

(3/4) Extend the code in (1) and (2) to include the power operator. This requires proper account of associativity of the operators. The power operator is right-associative, whereas the other operators are left-associative. Left-associative operators are popped off if the precedence is bigger or equal, while right-associative operators are only popped off if the precedence is bigger. [1 Marks]