# Coursework 9 (Scala)

This coursework is worth 10%. It is about a small programming language called brainf***. The first part is due on 13 December at 11pm; the second, more advanced part, is due on 20 December at 11pm.

**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.

## Part 1 (6 Marks)

Coming from Java or C++, you might think Scala is a rather esoteric programming language. But remember, some serious companies have built their business on Scala.[2] And there are far, far more esoteric languages out there. One

[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!

[2] https://en.wikipedia.org/wiki/Scala_(programming_language)#Companies

is called *brainf****. You are asked in this part to implement an interpreter and compiler for this language.

Urban Müller developed brainf*** in 1993. A close relative of this language was already introduced in 1964 by Corado Böhm, an Italian computer pioneer. The main feature of brainf*** is its minimalistic set of instructions—just 8 instructions in total and all of which are single characters. Despite the minimalism, this language has been shown to be Turing complete…if this doesn't ring any bell with you: it roughly means that every algorithm we know can, in principle, be implemented in brainf***. It just takes a lot of determination and quite a lot of memory resources. Some relatively sophisticated sample programs in brainf*** are given in the file `bf.scala`, including a brainf*** program for the Sierpinski triangle and Mandelbot set.

As mentioned above, brainf*** has 8 single-character commands, namely `'>'`, `'<'`, `'+'`, `'-'`, `'.'`, `','`, `'['` and `']'`. Every other character is considered a comment. Brainf*** operates on memory cells containing integers. For this it uses a single memory pointer that points at each stage to one memory cell. This pointer can be moved forward by one memory cell by using the command `'>'`, and backward by using `'<'`. The commands `'+'` and `'-'` increase, respectively decrease, by 1 the content of the memory cell to which the memory pointer currently points to. The commands for input/output are `','` and `'.'`. Output works by reading the content of the memory cell to which the memory pointer points to and printing it out as an ASCII character. Input works the other way, taking some user input and storing it in the cell to which the memory pointer points to. The commands `'['` and `']'` are looping constructs. Everything in between `'['` and `']'` is repeated until a counter (memory cell) reaches zero. A typical program in brainf*** looks as follows:

```
++++++++[>++++[>++>+++>+++>+<<<<-]>+>+>->>+[<]<-]>>.>---.+++++++
..+++.>>.<-.<.+++.------.--------.>>+.>++.
```

This one prints out Hello World…obviously.

**Tasks (file bf.scala)**

(1) Write a function that takes a file name as argument and and requests the corresponding file from disk. It returns the content of the file as a String. If the file does not exists, the function should return the empty string.
[1 Mark]

(2) Brainf*** memory is represented by a `Map` from integers to integers. The empty memory is represented by `Map()`, that is nothing is stored in the memory; `Map(0 -> 1, 2 -> 3)` stores `1` at memory location `0`, and at `2` it stores `3`. The convention is that if we query the memory at a location that is *not* defined in the `Map`, we return `0`. Write a function, `sread`, that takes a memory (a `Map`) and a memory pointer (an `Int`) as argument, and 'safely'

reads the corresponding memory location. If the `Map` is not defined at the memory pointer, `sread` returns `0`.

Write another function `write`, which takes a memory, a memory pointer and an integer value as argument and updates the `Map` with the value at the given memory location. As usual the `Map` is not updated 'in-place' but a new map is created with the same data, except the value is stored at the given memory pointer. [1 Mark]

(3) Write two functions, `jumpRight` and `jumpLeft` that are needed to implement the loop constructs of brainf***. They take a program (a `String`) and a program counter (an `Int`) as argument and move right (respectively left) in the string in order to find the **matching** opening/closing bracket. For example, given the following program with the program counter indicated by an arrow:

```
--[..+>--],>,++
   ↑
```

then the matching closing bracket is in 9th position (counting from 0) and `jumpRight` is supposed to return the position just after this

```
--[..+>--],>,++
          ↑
```

meaning it jumps to after the loop. Similarly, if you are in 8th position then `jumpLeft` is supposed to jump to just after the opening bracket (that is jumping to the beginning of the loop):

```
                     jumpLeft
--[..+>--],>,++      ------>      --[..+>--],>,++
        ↑                            ↑
```

Unfortunately we have to take into account that there might be other opening and closing brackets on the 'way' to find the matching bracket. For example in the brainf*** program

```
--[..[+>]--],>,++
   ↑
```

we do not want to return the index for the `'-'` in the 9th position, but the program counter for `','` in 12th position. The easiest to find out whether a bracket is matched is by using levels (which are the third argument in `jumpLeft` and `jumpLeft`). In case of `jumpRight` you increase the level by one whenever you find an opening bracket and decrease by one for a closing bracket. Then in `jumpRight` you are looking for the closing bracket on level `0`. For `jumpLeft` you do the opposite. In this way you can find **matching** brackets in strings such as

```
--[..[[-]+>[.]]--],>,++
   ↑
```

for which `jumpRight` should produce the position:

```
--[..[[-]+>[.]]--],>,++
                  ↑
```

It is also possible that the position returned by `jumpRight` or `jumpLeft` is outside the string in cases where there are no matching brackets. For example

```
--[..[[-]+>[.]]--,>,++   --jumpRight-->   --[..[[-]+>[.]]-->,++
   ↑                                                          ↑
```

[2 Marks]

(4) Write a recursive function `run` that executes a brainf*** program. It takes a program, a program counter, a memory pointer and a memory as arguments. If the program counter is outside the program string, the execution stops and `run` returns the memory. If the program counter is inside the string, it reads the corresponding character and updates the program counter `pc`, memory pointer `mp` and memory `mem` according to the rules shown in Figure 1. It then calls recursively `run` with the updated data. The most convenient way to implement the rules in `run` is to use pattern-matching and calculating a triple consisting of the new `pc`, `mp` and `mem`.

Write another function `start` that calls `run` with a given brainfu** program and memory, and the program counter and memory pointer set to 0. Like `run` it returns the memory after the execution of the program finishes. You can test your brainf**k interpreter with the Sierpinski triangle or the Hello world programs (they seem to be particularly useful for debugging purposes), or have a look at

https://esolangs.org/wiki/Brainfuck

[2 Marks]

## Part 2 (4 Marks)

While it is fun to look at bf-programs, like the Sierpinski triangle or the Mandelbrot program, being interpreted, it is much more fun to write a compiler for the bf-language.

| | |
|---|---|
| `'>'` | • `pc` + 1<br>• `mp` + 1<br>• `mem` unchanged |
| `'<'` | • `pc` + 1<br>• `mp` − 1<br>• `mem` unchanged |
| `'+'` | • `pc` + 1<br>• `mp` unchanged<br>• `mem` updated with `mp -> mem(mp) + 1` |
| `'-'` | • `pc` + 1<br>• `mp` unchanged<br>• `mem` updated with `mp -> mem(mp) - 1` |
| `'.'` | • `pc` + 1<br>• `mp` and `mem` unchanged<br>• print out `mem(mp)` as a character |
| `','` | • `pc` + 1<br>• `mp` unchanged<br>• `mem` updated with `mp ->` input<br>the input is given by `Console.in.read().toByte` |
| `'['` | if `mem(mp) == 0` then<br>• `pc = jumpRight(prog, pc + 1, 0)`<br>• `mp` and `mem` unchanged<br><br>otherwise if `mem(mp) != 0` then<br>• `pc` + 1<br>• `mp` and `mem` unchanged |
| `']'` | if `mem(mp) != 0` then<br>• `pc = jumpLeft(prog, pc - 1, 0)`<br>• `mp` and `mem` unchanged<br><br>otherwise if `mem(mp) == 0` then<br>• `pc` + 1<br>• `mp` and `mem` unchanged |
| any other char | • `pc` + 1<br>• `mp` and `mem` unchanged |

Figure 1: The rules for how commands in the brainf*** language update the program counter `pc`, memory pointer `mp` and memory `mem`.