

## Main Part 5 (Scala, 10 Marks)

*"If there's one feature that makes Scala, 'Scala',  
I would pick implicits."  
— Martin Odersky (creator of the Scala language)*

This part is about a small (esoteric) programming language called brainf\*\*\*. The task is to implement an interpreter and compiler for this language.

### Important

- Make sure the files you submit can be processed by just calling `scala-cli compile <<filename.scala>>` on the command line.<sup>1</sup> 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 `Arrays` or `ListBuffers`, 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 Copilot for help or do any other AI-shenanigans like this! An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

---


<sup>1</sup>All major OSes, including Windows, have a command line. So there is no good reason to not download `scala-cli`, install it and run it on your own computer. Just do it!

## Reference Implementation

As usual, this Scala assignment comes with a reference implementation in form of two `jar`-files. You can download them from KEATS. They allow you to run any test cases on your own computer. For example you can call `scala-cli` on the command line with the option `--extra-jars bf.jar` and then query any function from the `bf.scala` template file. You have to prefix the calls with `M5a` and `M5b`, respectively. For example

```
$ scala-cli --extra-jars bf.jar
scala> import M5a._
scala> run(load_bff("sierpinski.bf")) ; ()
```

## Part A (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.<sup>2</sup> I claim functional programming is not a fad. And there are far, far more esoteric languages out there. One is called *brainf\*\*\**. You are asked in this part to implement an interpreter for this language. 

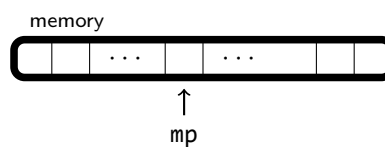
Urban Müller developed the original version of *brainf\*\*\** in 1993. A close relative of this language was already introduced in 1964 by Corrado Böhm, an Italian computer pioneer. The main feature of *brainf\*\*\** is its minimalistic set of instructions—just 8 instructions in total and all of which 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 can, in principle, be implemented in *brainf\*\*\**. It just takes a lot of determination and quite a lot of memory and time.

Some relatively sophisticated sample programs in *brainf\*\*\** are given in the file `bf.scala`, including a *brainf\*\*\** program for the Sierpinski triangle and the Mandelbrot set.<sup>3</sup> There seems to be even a dedicated Windows IDE for *bf* programs, though I am not sure whether this is just an elaborate April fools' joke—judge yourself:

<https://www.microsoft.com/en-us/p/brainf-ck/9nblgggzhvq5>


As mentioned above, the original *brainf\*\*\** has 8 single-character commands. Our version of *bf* will contain the commands '>', '<', '+', '-', '.', '[', and ']'. Every other character is considered a comment.

Our interpreter for *bf* operates on memory cells containing integers. For this it uses a single memory pointer, called *mp*, 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 command for output in *bf* is '.' whereby 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.<sup>4</sup> The commands '['

<sup>2</sup>[https://en.wikipedia.org/wiki/Scala\\_\(programming\\_language\)#Companies](https://en.wikipedia.org/wiki/Scala_(programming_language)#Companies)

<sup>3</sup>Of course somebody tried LLMs for writing *bf*-programs. As expected, they did spectacularly badly in this task. 

<sup>4</sup>In the original version of *bf*, there is also a command for input, but we omit it here. All our programs will be "autonomous".

and ']' are looping constructs. Everything in between '[' and ']' is repeated until a counter (memory cell) reaches zero. A typical program in brainf\*\*k looks as follows:

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

This one prints out Hello World...obviously ;o)

### Tasks (file bf.scala)

- (1) Write a function that takes a filename (a string) as an argument and requests the corresponding file from disk. It returns the content of the file as a string. If the file does not exist, the function should return the empty string. [0.5 Marks]
- (2) Brainf\*\*k 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 'safe-read' function, `sread`, that takes a memory (a Map) and a memory pointer (an Int) as arguments, 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 arguments 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 new value is stored at the given memory pointer. [0.5 Marks]
- (3) Write two functions, `jumpRight` and `jumpLeft`, that are needed to implement the loop constructs in brainf\*\*k. They take a program (a String) and a program counter (an Int) as arguments 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):

$$\begin{array}{ccc} \text{--}[\text{..+>--}]\text{.}>\text{.++} & \xrightarrow{\text{jumpLeft}} & \text{--}[\text{..+>--}]\text{.}>\text{.++} \\ \uparrow & & \uparrow \end{array}$$

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 `brain*ck` 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

$$\begin{array}{ccc} \text{--}[\text{.}[\text{[-]}+\text{>}[\text{.}]]\text{--}.\text{>}.++ & \xrightarrow{\text{jumpRight}} & \text{--}[\text{.}[\text{[-]}+\text{>}[\text{.}]]\text{--}.\text{>}.++ \\ \uparrow & & \uparrow \end{array}$$

[2 Marks]

- (4) Write a recursive function `compute` that runs a `brain*u*k` 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 `compute` 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 `compute` with the updated data. The most convenient way to implement the `brainfuck` rules in Scala is to use pattern-matching and to calculate a triple consisting of the updated `pc`, `mp` and `mem`.

Write another function `run` that calls `compute` with a given `brainfuck` program and memory, and the program counter and memory pointer set to 0. Like `compute`, it returns the memory after the execution of the program finishes. You can test your `brainfuck` 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>

for more bf-programs and the test cases given in `bf.scala`.

[2 Mark]

- (5) Let us also generate some BF programs ourselves: Write a function `generate` which takes a list of characters as input and generates a corresponding BF program that prints out this list of characters. One way to generate such a program is to consider each character in the list, add as many "+" given by the ASCII code of the character, then add the "." command for printing and add the loop "[ - ]" for "zero-ing" the memory cell; then go to the next character in the list. For example the list `"ABC".toString` produces (as a single string)

```
+++++
+++++. [-]+++++
+++++. [-]+++++
+++++. [-]
```

[1 Mark]

## Part B (4 Marks)

I am sure you agree while it is fun to marvel 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.

### Tasks (file `bfc.scala`)

- (6) Compilers, in general, attempt to make programs run faster by precomputing as much information as possible before running the program. In

'>'	<ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• <math>mp + 1</math></li> <li>• mem unchanged</li> </ul>
'<'	<ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• <math>mp - 1</math></li> <li>• mem unchanged</li> </ul>
'+'	<ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• mp unchanged</li> <li>• mem updated with <math>mp \rightarrow mem(mp) + 1</math></li> </ul>
'-'	<ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• mp unchanged</li> <li>• mem updated with <math>mp \rightarrow mem(mp) - 1</math></li> </ul>
'.'	<ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• mp and mem unchanged</li> <li>• print out <math>mem(mp)</math> as a character</li> </ul>
'['	<p>if <math>mem(mp) == 0</math> then</p> <ul style="list-style-type: none"> <li>• <math>pc = \text{jumpRight}(\text{prog}, pc + 1, 0)</math></li> <li>• mp and mem unchanged</li> </ul> <p>otherwise if <math>mem(mp) != 0</math> then</p> <ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• mp and mem unchanged</li> </ul>
']'	<p>if <math>mem(mp) != 0</math> then</p> <ul style="list-style-type: none"> <li>• <math>pc = \text{jumpLeft}(\text{prog}, pc - 1, 0)</math></li> <li>• mp and mem unchanged</li> </ul> <p>otherwise if <math>mem(mp) == 0</math> then</p> <ul style="list-style-type: none"> <li>• <math>pc + 1</math></li> <li>• mp and mem unchanged</li> </ul>
any other char	<ul style="list-style-type: none"> <li>• mp and mem unchanged</li> </ul>

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

our case we can precompute the addresses where we need to jump at the beginning and end of loops.

For this write a function `jtable` that precomputes the “jump table” for a bf-program. This function takes a bf-program as an argument and returns a `Map[Int, Int]`. The purpose of this Map is to record the information, in cases a pc-position points to a '[' or a ']', to which pc-position do we need to jump next?

For example for the program

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

we obtain the Map (note the precise numbers might differ depending on white spaces etc. in the bf-program):

```
Map(69 -> 61, 5 -> 20, 60 -> 70, 27 -> 44, 43 -> 28, 19 -> 6)
```

This Map states that for the '[' on position 5, we need to jump to position 20, which is just after the corresponding ']'. Similarly, for the ']' on position 19, we need to jump to position 6, which is just after the '[' on position 5, and so on. The idea is to not calculate this information each time we hit a bracket, but just look up this information in the `jtable`.

Then adapt the `compute` and `run` functions from Part 1 in order to take advantage of the information stored in the `jtable`. This means whenever `jumpLeft` and `jumpRight` was called previously, you should look up the jump address in the `jtable`. Feel free to reuse the function `jumpLeft` and `jumpRight` for calculating the `jtable`. [1 Mark]

- (7) Compilers try to eliminate any “dead” code that could slow down programs and also perform what is often called *peephole optimisations*.<sup>5</sup> For the latter consider that it is difficult for compilers to comprehend what is intended with whole programs, but they are very good at finding out what small snippets of code do, and then try to generate faster code for such snippets.

In our case, dead code is everything that is not a bf-command. Therefore write a function `optimise` which deletes such dead code from a bf-program. Moreover this function should replace every substring of the form `[-]` by a new command `0`. The idea is that the loop `[-]` just resets the memory at the current location to 0. It is more efficient to do this in a single step, rather than stepwise in a loop as in the original bf-programs.

In the extended `compute3` and `run3` functions you should implement this command by writing 0 to `mem(mp)`, that is use `write(mem, mp, 0)` as the

<sup>5</sup>[https://en.wikipedia.org/wiki/Peephole\\_optimization](https://en.wikipedia.org/wiki/Peephole_optimization)



rule for the command 0. The easiest way to modify a string in this way is to use the regular expression `""[<>+\\-\\.\\[\\]]""`, which recognises everything that is not a bf-command. Similarly, the regular expression `""\\[-\\]""` finds all occurrences of `[-]`. By using the Scala method `.replaceAll` you can replace substrings with new strings.

[1 Mark]

- (8) Finally, real compilers try to take advantage of modern CPUs which often provide complex operations in hardware that can combine many smaller instructions into a single faster instruction.

In our case we can optimise the several single increments performed at a memory cell, for example `++++`, by a single “increment by 4”. For this optimisation we just have to make sure these single increments are all next to each other. Similar optimisations should apply for the bf-commands `-`, `<` and `>`, which can all be replaced by extended versions that take the amount of the increment (decrement) into account. We will do this by introducing two-character bf-commands. For example

original bf-cmds	replacement
<code>+</code>	<code>+A</code>
<code>++</code>	<code>+B</code>
<code>+++</code>	<code>+C</code>
<code>...</code>	<code>...</code>
<code>+++...++</code>	<code>+Z</code>
(these are 26 +’s)	

If there are more than 26 +’s in a row, then more than one “two-character” bf-commands need to be generated (the idea is that more than 26 copies of a single bf-command in a row is a rare occurrence in actual bf-programs). Similar replacements apply for `-`, `<` and `>`, but all other bf-commands should be unaffected by this change.

For this write a function `combine` which replaces sequences of repeated increment and decrement commands by appropriate two-character commands. In the functions `compute4` and `run4`, the “combine” and the optimisation from (6) should be performed. Make sure that when a two-character bf-command is encountered you need to increase the pc-counter by two in order to progress to the next command. For example

```
combine(optimise(load_bff("benchmark.bf")))
```

generates the improved program

```
>A+B[<A+M>A-A]<A[[ ...
```

for the original benchmark program

```
>+{<+++++++>-}<[ ...
```

As you can see, the compiler bets on saving a lot of time on the +B and +M steps so that the optimisations is worthwhile overall (of course for the >A's and so on, the compiler incurs a penalty). Luckily, after you have performed all optimisations in (5) - (7), you can expect that the `benchmark.bf` program runs four to five times faster. You can also test whether your compiler produces the correct result by testing for example

```
run(load_bff("sierpinski.bf")) == run4(load_bff("sierpinski.bf"))
```

which should return true for all the different compiler stages.

[2 Marks]