Bonus Exercise: InfinityVM

General notes

- You have to implement the task in a functional way, i.e. without using side effects. Among other things, this means: don't use `var` variables, only `val`. Don't call methods that change state of anything. *Exception*: in the `run` method, input and output is allowed.
- You may use data structures and functions from the standard library, as long as they are functional (e.g. everything in `scala.collection.immutable`). You may also use the library `cats`, which is included in the sbt template.
- The task counts as passed, if your implementation passes all tests, the included sample program runs, and your code is written in a functional way.
- The tests are included in the template, you can run them locally. You can use your IDE / editor or run the command `test` in sbt.
- Submit your implementation by uploading your whole project folder as an archive (zip or tar.gz/tar.xz)

In this exercise you will implement a small virtual machine, the "Infinity VM". This machine has its own instruction set, the template source code contains a minigame you can play, when you have solved the exercise.

InfinityVM

The Infinity VM works similar to a normal computer, albeit very simplified. In each cycle, one "op code" is read from the machine's memory. This code represents a specific operation the machine will then execute. This could be e.g. setting a register or loading data from memory.

The Infinity VM has an infinite amount of registers (hence the name), numbered starting with 0. The registers basically are the memory on which calculations are executed. The VM also has an infinite amount of main memory, which is used for operations with longer data. Each register holds an `Int` and each index of the memory holds an `Int` as well. The VM uses a single main memory for data as well as program code. The state of our machine is held in a `case class`:

```
package infinityVM

final case class VMState(
  registers: Map[Int, Int],
  programCounter: Int,
  memory: Vector[Int],
  state: ExternalState,
)
```

So the VM has a map of registers. For each register it stores a value. Whenever a register is requested, for which there is no value stored in the map, the value 0 should be used.

It also has a `programCounter`, which tells uns from which address in memory the next instruction should be loaded. When starting the VM, this counter is 0 and therefore points to the first address in memory.

Then there is the main `memory`. It consists of a series of `Int`s and is infinitely large, which our data structure `Vector` isn't of course. So we need to simulate it. When data is written to an address outside the Vector's

size, it should be expanded by filling it with zeros until it is large enough. When reading from an address outside its size, the result should be 0. Suppose we have the following memory:

```
Vector(1, 2, 3, 4)</pre>
```

A read request for address 0 should return 1, for address 3 it should return 4. If address 10 is requested, 0 should be returned. If a 1 is written to address 10, the memory should look like this afterwards:

```
Vector(1, 2, 3, 4, 0, 0, 0, 0, 0, 0, 1)
```

Finally, there's the `state` attribute. With this attribute, the virtual machine can give information to the outside world. There are four possible values:

```
package infinityVM

enum ExternalState:
  case ReadRequested(address: Int)
  case WriteRequested(address: Int)
  case Halted
  case Running
```

These have the following meaning:

`ReadRequested(address)`	The machine waits for a user input, which is written to the specified address in main memory.
`WriteRequested(address)`	The machine wants to output a string. The string starts at the given address and ends at the first 0 after that address in memory.
`Halted`	The machine has stopped. The program can now exit.
`Running`	The machine is ready to execute the next operation.

So much for the introduction, now the exercise:

From integers to instructions

The machine supports the following instructions, which can be executed. Every instruction is made up of 4 integers. The first integer is the op-code, which tells us which type of instruction it is. The other three integers are the operands and may represent a register, an address in memory or a value. Not every instruction uses all three operands. If an instruction uses fewer operands, the remaining operands are ignored, but the instructions are still considered 4 ints long.

Op-code	Label	Operand 1	Operand 2	Operand 3	Description
00	HALT	•	•	•	Stops the machine by setting the state attribute accordingly.
01	Interrup-tRead	read address	•	•	Asks the machine to request input from the user and writing it to the specified address. More info below.
02	Inter-ruptWrite	write address	•	•	Requests the machine to display the string at the given address in memory to the user. More info below.
03	Zero	register	•	•	Sets the value of the given register to 0.
04	AddI	register	Wert	•	Adds the given value to the value of the given register and stores the value there.
05	Eql	comparison register1	comparison register2	result register	If the value of the first register is greater than the one in the second register, write 1 to the result register. If the values are the the same, then 0. otherwise -1.
06	Jeq	comparison register1	comparison register2	jump address	If the values of the given registers are the same, jump to the given address (i.e. set the `programCounter`).
07	Jump	jump address	•	•	Jump to the given address.
08	Load	data register	address register	•	Read the memory at the given address (value of the address register) and store the value in the data register.
09	Add	Register1	Register2	result register	Add the values in the first two registers together and store the result in the result register. Attention: the result register may be the same as one of the other two registers.
10	Mod	Register1	Register2	result register	Calculate (Register1 % Register2) and store the value in the result register. Attention: the result register may be the same as one of the other two registers.
11	Div	Register1	Register2	result register	Divide the value in register 1 by the one in register 2 and store it in the result register. Attention: the result register may be the same as one of the other two registers.
12	Store	data register	address register	•	Write the data from the data register to the given memory address (value of the address register)
13	NOP	•	•	•	Do nothing.

`def decodeInstruction(a: Int, b: Int, c: Int, d: Int): Instruction`

Implement the function specified above in `InfinityVM.scala`. It should accept four integers and turn them into the correct instruction from the `Instruction` enum. Should `a` not be a valid opcode, return `NOP`.

Executing the instructions

Now that we can map from integers to instructions, we want to execute those instructions. Our goal here is to implement a function, that takes the current state of the VM, reads 4 integers from `memory` at the position stored in `programCounter` in the machine state, decodes them into an instruction and executes it.

As we cannot change an existing state in functional programming, we return the new state instead. The `copy` method, which every case class provides, may be helpful here. You can see from the above table, what should be done with each instruction.

Note that after executing an instruction, except `Halt`, `Jeq` and `Jump`, the `programmCounter` has to be incremented by the size of an instruction. In case of `Jump` it is set to the jump address instead. For `Jeq` the comparison decides if the jump address is used, if not it increments like other commands. In case of `Halt`, the `programCounter` doesn't change.

Also note, that the instructions `HALT`, `InterruptRead` and `InterruptWrite` only set the `state` attribute and update the `programmCounter` if applicable, but don't do anything else. The actual handling of input and output happens outside the method.

`def executeInstruction(vm: VMState): VMState`

Implement the function above in the file `InfinityVM.scala`. It takes a VMState, of which it should return a copy modified as described above as the new state of the VM.

`@tailrec def run(vm: VMState): VMState`

Finally, implement the `run` method in the file `Main.scala`, which should be tail-recursive. In functional programming, we try to move side effects, if they can't be avoided, as far to the outer end of our program as possible. As reading user input and writing text to the screen are side effects, we banish those to the `run` method.

The `run` method's purpose is to run a whole program - not only a single instruction - on our VM. We pass it a starting state as input. The program to run is contained in `memory`. The virtual machine acts differently according to its current value of `state`:

- If the machine is in state `Running`, the current instruction should be executed. Then the simulation should continue running with the new state.
- If the machine is in state `Halted`, no further instruction shall be executed. The current VMState is returned unmodified.
- If the machine's state is `ReadRequested(addr)`, a line of text should be read from standard input. We assume here, that input only contains ASCII characters. These characters have to be converted to integers and written to the address `addr` in the VM's memory. Existing data at that location is overridden. Also add an additional 0 integer at the end, so that the program code running on the VM knows where the string ends. Example:

```
ReadRequested(4), Input: "abc"
in ASCII: [97, 98, 99]           target address
                                 ↓
Memory before: [ 11, 12, 13, 14, 15, 16, 17, 18, 19]

Memory after:  [ 11, 12, 13, 14, 97, 98, 99,  0, 19]
                                |--------------| Input string
```

Afterwards, the VM's `state` has to be set to `Running` again. The simulation shoud then continue from that point. For reading a line of input, you may use `scala.io.StdIn.readLine()`.

- If the machine's state is `WriteRequested(addr)`, a line of text should be written to standard output. For that, integers should be read from the VM's `memory` starting at address `addr` until the first 0 integer (exclusive). These integers shall be assumed to be ASCII values and made into a String (hint: a collection of `Char`s can be made into a string with `.mkString`). This string is then printed to standard output. Finally, set the `state` back to `Running` running and continue the simulation.

If you implemented all those methods correctly, you should be able to play a small puzzle game when starting the main method given in the template, in which you and the computer take turns in picking up coins.