Bonusaufgabe: InfinityVM

Allgemeine Hinweise

- Die Aufgabe muss funktional, das bedeutet ohne Seiteneffekte geschrieben werden. Unter anderem bedeutet das: Benutzen Sie keine `var` Variablen, nur `val` Variablen. Rufen Sie außerdem keine Methoden auf, die den State von irgendetwas verändern. *Ausnahme*: in der `run`-Methode dürfen Input und Output stattfinden, siehe dort.
- Sie dürfen Datenstrukturen und Funktionen aus der Standard-Library nutzen, sofern diese funktional sind (z.B. alles unter `scala.collection.immutable`). Außerdem dürfen Sie die Library `cats` nutzen, diese ist in der SBT-Vorlage eingebunden.
- Die Aufgabe gilt als bestanden, wenn Sie alle Tests erfolgreich bestehen, das enthaltene Beispielprogramm läuft und der Code funktional ist.
- Die Tests sind in der Vorlage enthalten, Sie können diese also lokal ausführen. Hierzu können Sie ihre IDE / Ihren Editor nutzen, oder in sbt den Befehl `test` verwenden.
- Zur Abgabe laden Sie Ihren Projektordner als Archiv (zip oder tar.gz/tar.xz) hoch.

InfinityVM

In dieser Aufgabe sollen Sie eine kleine virtuelle Maschine implementieren, die "Infinity VM". Diese Maschine hat ihren eigenen Befehlssatz und im Source-Code ist ein Mini-Spiel hinterlegt, dass Sie spielen können, wenn Sie die Aufgabe gelöst haben.

Die Infinity VM verhält sich wie ein normaler, stark vereinfachter Rechner. In jedem Takt wird ein "Op-Code" aus dem Speicher gelesen. Dieser Code steht für eine bestimmte Operation, welche die Maschine dann ausführt. Dabei kann es sich zum Beispiel darum handeln, ein Register zu setzen oder Daten aus dem Speicher zu laden.

Die Infinity VM hat unendliche viele Register (daher der Name), einfach durchnummeriert von 0 an. Die Register sind quasi der Speicher, auf dem direkt Rechenoperationen ausgeführt werden. Die VM besitzt außerdem unendlich viel Hauptspeicher, der für Operationen mit längeren Daten genutzt wird. In jedem der Register steckt ein `Int`, genauso wie an jeder Speicherstelle ein `Int` steckt. Sie besitzt nur einen Speicher, in dem sowohl Daten als auch Programmcode liegt. Der Zustand der ganzen Maschine ist in einer `case class` gehalten:

```
package infinityVM

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

Die VM hält also zunächst eine Map mit Registern. Für jedes dieser Register hält sie einen Wert. Prinzipiell gilt: Sollte der Wert eines Registers abgefragt werden, das Register jedoch nicht in der Map sein, so soll einfach 0 als Wert benutzt werden.

Außerdem hält die VM einen `programCounter` welcher angibt, von welcher Speicheradresse die nächste Instruktion geladen werden soll. Zu Beginn ist dieser Counter 0 und zeigt damit auf die erste Adresse im Speicher.

Weiterhin gibt es natürlich den Speicher der Maschine. Dieser ist im `memory` Attribute gehalten. Der Speicher besteht aus lauter `Int`s. Er ist unendlich groß, was unsere Datenstruktur `Vector` natürlich nicht ist, wir müssen das also simulieren. Wenn Daten an einen Stelle geschrieben werden, die außerhalb der Größe des Vectors liegt, soll er erweitert werden, indem er mit 0 aufgefüllt wird. Wird von einer Stelle außerhalb seinen Größe gelesen, soll 0 zurück gegeben werden. Nehmen wir an wir haben den folgenden Speicher:

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

Eine Speicheranfrage an die Stelle 0 soll 1 und an die Stelle 3 soll 4 zurück geben. Wenn nach der Stelle 10 gefragt wird, soll 0 zurück geben werde. Wird eine 1 an die Stelle 10 geschrieben, soll der Speicher danach so aussehen:

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

Zu guter Letzt gibt es noch das `state` Attribut. Mit diesem Attribut kann die virtuelle Maschine eine Information an die Aussenwelt geben. Mögliche Werte sind:

```
package infinityVM

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

Diese Werte bedeuten folgendes:

`ReadRequested(address)`	Die Maschine wartet darauf, dass eine Eingabe getätigt wird, die an die entsprechende Adresse im Speicher geschrieben werden soll.
`WriteRequested(address)`	Die Maschine will, dass ein String ausgegeben wird. Der String startet an der angegeben Adresse und endet an der ersten 0 nach dieser im Speicher.
`Halted`	Die Maschine hat gehalten. Das Gesamtprogramm kann nun beendet werden.
`Running`	Die Maschine wartet darauf, dass sie die nächste Operation ausführen darf.

So viel als Einleitung, nun beginnt die Aufgabe:

Von Integern zu Instruktionen

Die Maschine unterstützt die folgenden Instruktionen, welche ausgeführt werden können. Jede Instruktion besteht immer aus 4 Integern. Der erste Integer ist immer der Op-Code, welcher angibt, um welchen Typ von Instruktion es sich handelt. Die anderen drei Integer geben die Operanden an und können für ein Register, eine Adresse im Speicher oder einen Wert stehen. Nicht jede Instruktion nutzt auch alle 3 Operanden. Wenn nicht alle Operanden genutzt werden, werden die anderen Operanden einfach ignoriert. Die Instruktion gilt aber trotzdem immer als 4 Integer lang.

Op-code	Bezeichnung	Operand 1	Operand 2	Operand 3	Beschreibung
00	HALT	•	•	•	Hält die Maschine an, indem sie das state-Attribute entsprechend setzt.
01	InterruptRead	Leseadresse	•	•	Bittet darum, dass ein Nutzer Text eingibt und dieser an die angegeben Adresse gespeichert wird. Unten mehr dazu.
02	InterruptWrite	Schreibadresse	•	•	Bittet darum, dass der Text an der angegebenen Adresse dem Nutzer angezeigt wird. Unten mehr dazu.
03	Zero	Register	•	•	Setzt den Wert des angegebenen Registers auf 0.
04	AddI	Register	Wert	•	Addiert den angegebenen Wert auf den Wert des angegebenen Registers und speichert den Wert dort.
05	Eql	Vergleichsregister1	Vergleichsregistger2	Ergebnisregister	Ist der Wert des ersten Registers größer als der des zweiten Registers, so soll 1 in das Ergebnisregister geschrieben werden. Sind die Werte gleich, dann 0. Ansonsten -1.
06	Jeq	Vergleichsregister1	Vergleichsregistger2	Sprungadresse	Sind die Werte der beiden Register gleich, so soll an die angegebene Adresse gesprungen werden (d.h. der `programCounter` gesetzt werden)
07	Jump	Sprungadresse	•	•	Springe zur angegebenen Adresse.
08	Load	Datenregister	Adressregister	•	Lese den Speicher an der angegebenen Adresse (Wert des Adressregisters) aus und speichere den Wert im Datenregister.
09	Add	Register1	Register2	Ergebnisregister	Addiere die Register auf und speichere den Wert im Ergebnisregister. Achtung, das Ergebnisregister kann das gleiche wie eines der anderen beiden Register sein.
10	Mod	Register1	Register2	Ergebnisregister	Berechne (Register1 % Register2) und speichere den Wert im Ergebnisregister. Achtung, das Ergebnisregister kann das gleiche wie eines der anderen beiden Register sein.
11	Div	Register1	Register2	Ergebnisregister	Teile Register1 durch Register2 und speichere den Wert im Ergebnisregister. Achtung, das Ergebnisregister kann das gleiche wie eines der anderen beiden Register sein.
12	Store	Datenregister	Adressregister	•	Schreibe die Daten aus dem Datenregister an die angegebenen Speicheradresse (Wert des Adressregisters)
13	NOP	•	•	•	Tue nichts.

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

Implementieren Sie die oben angegebene Funktion in `InfinityVM.scala`. Sie soll 4 Integer entgegen nehmen und auf die korrekte Instruktion aus dem `Instruction`-Enum abbilden. Sollte es sich bei `a` nicht um einen validen Op-Code handeln, dann geben Sie `NOP` zurück.

Ausführen der Instruktionen

Nachdem wir von Integern auf Instruktionen mappen können, wollen wir die Instruktionen jetzt ausführen. Hierbei geht es darum, dass man eine Funktion implementiert, die den jetzigen Zustand einer VM entgegen nimmt, sich die ersten 4 Integer an der im Zustand als `programmCounter` angegebenen Stelle des `memory` holt, diese dekodiert und dann ausführt. Weil wir Zustand in FP nie ändern können, geben wir stattdessen einen neuen Zustand zurück. Hier kann die `copy`-Methode hilfreich sein, die für jede case class existiert. Was für die jeweilige Instruktion zu tun ist, können Sie aus der obigen Tabelle ablesen.

Beachten Sie, dass nach Ausführung aller Instruktionen, abgesehen von `Halt`, `Jeq` und `Jump` der `programmCounter` um die Größe einer Instruktion erhöht werden muss. Bei `Jump` wird er stattdessen auf die angegebene Sprungadresse gesetzt, bei `Jeq` hängt es vom Vergleich ab, ob die Sprungadresse genutzt wird oder wie bei anderen Befehlen inkrementiert. Bei `Halt` ändert sich der Zähler nicht.

Beachten Sie außerdem, dass die Instruktionen `HALT`, `InterruptRead` und `InterruptWrite` nur das `state`-Attribute setzen und ggf. den `programmCounter` anpassen, aber sonst nichts tun. Die eigentliche Abhandlung von Ein- und Ausgabe findet außerhalb der Methode statt.

`def executeInstruction(vm: VMState): VMState`

Implementieren sie die oben angegebene Funktion in der `InfinityVM.scala` Datei. Sie erhält einen Zustand, soll den Zustand wie oben beschrieben Verändern und einen neuen Zustand zurück liefern.

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

Implementiere Sie nun zu guter Letzt die `run`-Methode in der `Main.scala` Datei, welche tail-rekursiv sein soll. In der funktionalen Programmierung versuchen wir Seiteneffekte, wenn Sie nicht vermeidbar sind, so weit an die Enden unseres Programms wie möglich zu schieben. Nachdem das einlesen von Tastatureingaben und das ausgeben von Text auf dem Bildschirm Seiteneffekte sind, haben wir diese in die `run`-Methode verbannt.

Die `run`-Methode ist nun dafür da, ein komplettes Programm - und nicht nur ein einzelne Instruktion - in unserer VM laufen zulassen. Sie bekommt den Startzustand der Maschine als Eingabe. Das auszuführende Programm wird hierbei im `memory` übergeben. Je nachdem in welchem Zustand die Maschine gerade ist, verhält sie sich unterschiedlich:

- Ist die Maschine im Zustand `Running`, so soll auf der Maschine die gerade vorliegende Instruktion ausgeführt werden. Dann soll die Simulation mit dem neuen Zustand weiterlaufen.
- Ist die Maschine im Zustand `Halted`, so soll keine Instruktion mehr ausgeführt werden. Der Zustand der VM wird einfach zurück gegeben.
- Ist die Maschine im Zustand `ReadRequested(addr)`, so soll eine Zeile von der Standard-Eingabe gelesen werden. Wir gehen davon aus, dass der Text nur ASCII-konforme Zeichen enthält. Diese Zeichen müssen in Integer umgewandelt werden und an die die Adresse `addr` an den Speicher der VM geschrieben werden. Vorhandene Daten werden dabei überschrieben. Fügen sie als letzte Stelle noch den 0-Integer an, damit der eventuelle Programm Code in der VM weiß, wann der String zu Ende ist. Beispiel:

```
ReadRequested(4), Eingabe: "abc"
in ASCII: [97, 98, 99]              Zieladresse
                                    ↓
Speicher vorher : [ 11, 12, 13, 14, 15, 16, 17, 18, 19]

Speicher nachher: [ 11, 12, 13, 14, 97, 98, 99,  0, 19]
```

```
                                    |--------------| Eingabestring
```

Dann muss der `state` der VM wieder auf `Running` gesetzt werden. Ab dieser Stelle soll die Simulation dann fortgesetzt werden. Für das Einlesen einer Zeile können Sie `scala.io.StdIn.readLine()` benutzen.

- Ist die Maschine im Zustand `WriteRequested(addr)`, so soll eine Zeile auf der Standard-Ausgabe ausgegeben werden. Dabei soll im `memory` der Maschine ab Adresse `addr` bis zum ersten 0-Integer(ausschließlich) gelesen werden. Diese Integer sollen als ASCII-Werte angenommen und in einen String gepackt werden (Hinweis: Eine Collection von `Char`s kann mit `.mkString` zu einem String gewandelt werden). `Dieser String wird dann auf der Standard-Ausgabe ausgegeben. Danach soll der`state`wieder auf`Running‘ gesetzt werden und die Simulation soll fortfahren.

Wenn Sie all das korrekt implementiert haben, sollte Sie beim Starten der vorgegebenen Main ein kleines Spiel spielen können, in dem Sie und der Computer abwechseln Münzen nehmen.