Universität der Bundeswehr
Fakultät für Informatik – Institut 2
Prof. Dr. Wolfgang Hommel
Michael Grabatin, Michael Steinke

WT 2020
Übungsblatt 3

Einführung in die Informatik II

31.01 und 03.02.2020

1 IBAN Verifikation

In der Vorlesung (Kap. 2, Folie 5) wurde gezeigt, wie die Gültigkeit einer Kontonummer mit Hilfe einer Prüfsumme überprüft werden kann. Diese Aufgabe soll nun eine Validierung für IBANs geschrieben werden.

Eine IBAN besteht aus:

- 2-stelliger Ländercode (Großbuchstaben)
- 2-stellige Prüfziffer (Ziffern)
- max. 30-stellige Kontonummer (Großbuchstaben oder Ziffern)

Um eine IBAN zu modelieren, soll in dieser Aufgabe folgender Typ verwendet werden.

```
1 type IBAN = Array[Char]
```

Um eine IBAN auf ihre Gültigkeit zu prüfen, wird diese in eine Zahl umgewandelt und diese $mod97$ genommen. Wenn die IBAN gültig ist, muss das Ergebnis 1 sein. Die Umwandlung in eine Zahl erfolgt folgendermaßen:

- Die ersten vier Zeichen (Ländercode und Prüfziffer) werden an das Ende der Kontonummer verschoben.
- Die in der IBAN enthaltenen Buchstaben werden entsprechend ihrer Position im Alphabet + 10 in Ziffern umgewandelt. Also $A = 10$, $B = 11$, ..., $Z = 35$.
- Die so erstellte, nur aus Ziffern bestehende, Zeichenkette wird in einen Integer umgewandelt, der durch $mod97$ geprüft werden kann.

Im Folgenden soll die Validierungsfunktion Schritt für Schritt entwickelt werden.

a) Vervollständigen Sie folgende Funktion, die die ersten 4 Zeichen einer übergebenen IBAN an das Ende verschiebt.

```
1  def rearrange(iban: IBAN): Array[Char] = {
2      if(iban.length < 5) {
3          // Damit die IBAN den Ländercode, Prüfziffer und eine
               Kontonummer enthält muss sie mindestens 5 Stellen haben.
4          // Ergänzen Sie hier die Fehlerbehandlung und lösen Sie eine
               Exception vom Typ Exception aus.
5
6      }
7      var rearranged = new Array[Char](iban.length) //Kopie des IBAN-
           Arrays zum modifizieren
8      // Verschieben der ersten 4 Zeichen
9      // ...
10
11     return rearranged
12 }
```

Hinweis: Zum Verschieben können Sie die in der Vorlesung vorgestellte Funktion `Array.copy` verwenden.

```
def rearrange(iban: IBAN): Array[Char] = {
    if(iban.length < 5) {
        throw new Exception("IBAN ist zu kurz")
    }
    var rearranged = new Array[Char](iban.length)
    Array.copy(iban, 4, rearranged, 0, iban.length-4)
    for(i <- 0 to 3) {
        rearranged(rearranged.length-4+i) = iban(i)
    }
    return rearranged
}
```

b) Implementieren Sie nun eine Funktion, die alle in einem `Array[Char]` vorkommenden Buchstaben, entsprechend des in der Aufgabenstellung gegebenen Schemas (Position im Alphabet + 10, $A = 10$, ..., $Z = 35$) umwandelt.

Bei einem `Char` kann über die Funktionen `.isDigit` oder `.isLetter` überprüft werden, ob das Zeichen eine Zahl oder ein Buchstabe ist. Außerdem kann ein `Char`, der ein Buchstabe ist, mit der Funktion `.getNumericValue` wie für diese Aufgabe benötigt, in ein Zahl umgewandelt werden:

```
'A'.getNumericValue = 10 : Int
```

Um aus dem `Array[Char]` einen String zu erzeugen können Sie die Funktion `.mkString` verwenden. Den String können Sie dann wie folgt in einen `BigInt` umwandeln. Ein `BigInt` ist hier notwendig, da die aus der IBAN resultierende Zahl zu groß für einen normalen Integer sein kann.

```
BigInt(Array[Char]('1','2').mkString)
```

Beim Erstellen des `BigInt` kann eine `NumberFormatException` ausgelöst werden, falls der gegebene String nicht in einen Integer umgewandelt werden kann. Verwenden Sie einen `try`, `catch` Block um diese Exception abzufangen und lösen sie in dem `catch`-Teil eine eigene Exception zur Fehlerbehandlung aus.

Füllen Sie im gegebenen Funktionsgerüst die mit dem Platzhalter `???` markierten Stellen.

```
def convert2int(sa: Array[Char]): BigInt = {
  var lc = 0 // letter count

  ???

  // Neues Array mit Platz für die
  // expandierten Buchstaben
  var converted = new Array[Char](sa.length+lc)

  ???
  for(i <- 0 until sa.length) {
    ???
  }

  ???
  return BigInt(converted.mkString)
  ???
}
```

Eine mögliche Implementierung ist:

```
def convert2int(sa: Array[Char]): BigInt = {
    var lc = 0
    for(c <- sa) {
        if(c.isLetter) {
            lc += 1
        }
    }

    var converted = new Array[Char](sa.length+cc)

    var skip = 0
    for(i <- 0 until sa.length) {
        if(sa(i).isDigit) {
            converted(i+skip) = sa(i)
        } else if(sa(i).isLetter) {
            var tmp = sa(i).getNumericValue.toString
            converted(i+skip) = tmp(0)
            skip += 1
            converted(i+skip) = tmp(1)
        } else {
            throw new Exception(s"Unerwarteter Wert ${sa(i)}")
        }
    }

    try {
        return BigInt(converted.mkString)
    } catch {
        case e:NumberFormatException => throw new Exception(s"
           Konnte die Zeichenkette '${sa.mkString}' nicht in
           einen Integer umwandeln: '${converted.mkString}'")
    }
}
```

c) Vervollständigen Sie nun die Funktion, die eine IBAN überprüft und dabei im Fehlerfall die Exceptions abfängt und deren Nachricht als Text ausgibt.

```
var iban1 : IBAN = Array[Char]('G', 'B', '8', '2', 'W', 'E', 'S', 'T'
    , '1', '2', '3', '4', '5', '6', '9', '8', '7', '6', '5', '4', '3',
     '2')
var iban2 : IBAN = Array[Char]('D', 'E', '1', '9', '1', '2', '3', '4'
    , '1', '2', '3', '4', '1', '2', '3', '4', '1', '2', '3', '4', '1',
     '2')

def verify(iban: IBAN): Boolean = {
  ???
  try {
    ???
  } catch {
    ???
  }
  ???
}
```

Eine mögliche Implementierung ist:

```
1  var iban1 : IBAN = Array[Char]('G', 'B', '8', '2', 'W', 'E', 'S',
       'T', '1', '2', '3', '4', '5', '6', '9', '8', '7', '6', '5', '4
       ', '3', '2')
2  var iban2 : IBAN = Array[Char]('D', 'E', '1', '9', '1', '2', '3',
       '4', '1', '2', '3', '4', '1', '2', '3', '4', '1', '2', '3', '4
       ', '1', '2')
3
4  def verify(iban: IBAN): Boolean = {
5    var r: Array[Char] = new Array[Char](iban.length)
6    var i: BigInt = BigInt(0)
7
8    try {
9      r = rearrange(iban)
10     i = convert2int(r)
11   } catch {
12     case e: Exception => {
13       println(e.getMessage)
14       return false
15     }
16   }
17
18   return i % 97 == 1
19 }
```

2 Sudoku

In dieser Aufgabe sollen einzelne Funktionen für Sudoku-Arrays (Vorlesung Kap. 1, Folie 49) implementiert werden. Ein Sudoku-Array ist wie folgt definiert:

```
1 type Sudoku = Array[Array[Int]]
```

Hinweis: Anders als herkömmliche Sudokus werden die Elemente der Sudoku-Arrays mit den Zahlen von 0 bis 8 (und **nicht** von 1 bis 9) belegt.

a) Vervollständigen Sie folgende Funktion, die feststellt, ob zwei Sudoku-Arrays den gleichen Inhalt haben.

```
1 def gleich(s1: Sudoku, s2: Sudoku): Boolean = ...
```

Hinweis: Sie dürfen voraussetzen, dass beide Parameter 9 Zeilen mit je 9 Werten enthalten.

> Bei folgender Implementierung wird Element für Element auf Gleichheit getestet:
>
> ```
> 1 def gleich(s1: Sudoku, s2: Sudoku): Boolean = {
> 2     for (r <- 0 to 8) {
> 3       for (c <- 0 to 8) {
> 4         if (s1(r)(c) != s2(r)(c)) return false
> 5       }
> 6     }
> 7     return true
> 8 }
> ```

b) Sei P eine Permutation von $0, 1, 2, ..., 8$ und sei S ein Sudoku-Array. Schreiben Sie eine Funktion `wendeAuf`, so dass der Aufruf `wendeAuf(P, S)` einen Sudoku-Array zurückgibt, der sich durch Anwendung von P auf jedes Element von S ergibt. Für welches Q ergibt `wendeAuf(Q, S)` einen Sudoku-Array, dessen erste Zeile `Array(0,1,2,3,4,5,6,7,8)` ist?

> Eine mögliche Implementierung wäre wie folgt:
>
> ```
> 1 def wendeAuf(p: Perm, s: Sudoku): Sudoku = {
> 2     Array.tabulate(9, 9)((r, c) => p(s(r)(c)))
> 3 }
> ```
>
> Betrachtet man die erste Zeile als Permutation und bildet die zu ihr inverse Permutation (Vorlesung Kap. 1, Folie 48), so erhält man die gesuchte Permutation Q.

c) Aus einer einzigen Sudoku-Lösung entstehen durch Anwenden von Permutationen 9! Sudoku-Lösungen. Wir wollen nun feststellen, ob sich eine Sudoku-Lösung $S2$ aus einer Sudoku-Lösung $S1$ durch Anwenden einer Permutation ergibt, d.h. ob es eine Permutation $PS1S2$ gibt mit
`gleich(S2, wendeAuf(PS1S2, S1))` ergibt „*true*“.

Um die Frage zu beantworten, ermitteln wir zunächst die (einzige!) Permutation Q, welche die erste Zeile von $S1$ in die erste Zeile von $S2$ überführt. Gilt für Q, dass

```
1 gleich(S2, wendeAuf(Q, S1))
```

dann ist die Frage mit „ja“ zu beantworten, sonst mit „nein“. Implementieren Sie dieses Entscheidungsverfahren als Scala-Funktion.

Tipp: Die Anwendung von Permutationen, wie durch die Funktion `wendeAn` definiert (Vorlesung Kap. 1, Folie 47), verhält sich assoziativ: Sei $\circ$ die Abbildung, die eine Permutation gemäß der Funktion `wendeAn` auf eine andere anwendet und seien P, Q und R Permutationen, dann gilt $(P \circ Q) \circ R = P \circ (Q \circ R)$.

Vorüberlegung: Wie findet man zu zwei gegebenen Permutationen P und Q die Permutation R, die durch Anwendung auf P Q erzeugt?

Man mache sich zunächst deutlich, dass man Q erhält, wenn man zuerst die inverse Permutation von P (*hier:* P^{-1}) auf P und auf das Ergebnis anschließend Q anwendet, also $Q = Q \circ (P^{-1} \circ P)$. Aufgrund der Assoziativität gilt $Q = (Q \circ P^{-1}) \circ P$. Damit ist $R = Q \circ P^{-1}$ die gesuchte Permutation die P auf Q abbildet.

Mit den bekannten Funktionen aus der Vorlesung, lässt sich die Permutation $PS1S2$, welche (zunächst) die erste Zeile einer Sudoku-Lösung $S1$ auf die erste Zeile einer zweiten Sudoku-Lösung $S2$ abbildet, durch folgenden Aufruf ermitteln:

```
1 val PS1S2 = wendeAn(S2(0), invers(S1(0)))
```

Die Implementierung lässt sich daher wie folgt realisieren:

```
1 def existiertPerm(s1: Sudoku, s2: Sudoku): Boolean = {
2   val PS1S2 = wendeAn(s2(0), invers(s1(0)))
3
4   return gleich(wendeAuf(PS1S2, s1), s2)
5 }
```

d) Beim Sudoku gibt es neben dem Sudoku-Array einen Array `kandidaten` (Vorlesung Kap. 1, Folie 52).

- Welches Element von `kandidaten` muss „$true$“ sein, damit die Zuweisung

  ```
  1 sudoku(i)(j) = k     //(*)
  ```

  erlaubt ist?
- Welche Elemente von `kandidaten` sind im Anschluss an diese Zuweisung auf „$false$“ zu setzen?

Vervollständigen Sie die folgende Funktion, die genau dann „$false$“ zurückgibt, wenn die Zuweisung `(*)` nicht zulässig ist; anderenfalls wird an den globalen Arrays `sudoku` die in den Spiegelstrichen beschriebenen Änderungen vorgenommen (und „$true$“ zurückgegeben).

```
1 def setze(i: Int, j: Int, k: Int): Boolean = ...
```

Der Array `kandidaten` hält für jedes Feld eine charakteristische Funktion erlaubter Kandidaten.

- Damit das Feld (i, j) gesetzt werden kann, muss `kandidaten(i)(j)(k)` „$true$“ sein.
- Beim Besetzen des Feldes (i, j) mit dem Wert k müssen im Array `kandidaten` alle Elemente in der Spalte j, alle Elemente in der Reihe i sowie alle Elemente im entsprechenden Block auf „$false$“ gesetzt werden. Die erste Reihe und erste Spalte des jeweiligen Blocks lassen sich durch folgende Aufrufe bestimmen:

  ```
  1 val rb = (i / 3) * 3 //erste Reihe
  2 val cb = (j / 3) * 3 //erste Spalte
  ```

  Hinweis: Bei `i / 3` und `j / 3` handelt es sich um ganzzahlige Divisionen, daher ist `(i / 3) * 3` nur dann `i`, wenn `i` durch drei teilbar ist.

Damit lässt sich folgende Implementierung realisieren:

```
1 def setze(i: Int, j: Int, k: Int): Boolean = {
2     if (!kandidaten(i)(j)(k)) return false
3
4     //Erste Reihe / Spalte des Blocks
5     val rb = (i / 3) * 3
6     val cb = (j / 3) * 3
```

```
7
8       for (dr <- 0 to 2) {
9           for (dc <- 0 to 2) {
10              kandidaten(rb + dr)(cb + dc)(k) = false
11          }
12      }
13
14      //Spalte
15      for (c <- 0 to 8) kandidaten(i)(c)(k) = false
16
17      //Zeile
18      for (r <- 0 to 8) kandidaten(r)(j)(k) = false
19
20      //Wert setzen
21      sudoku(i)(j) = k
22      return true
23  }
```

3 Zwiebelringmatrix

In dieser Aufgabe soll eine quadratische Matrix mit der Kantenlänge n spiralförmig von 1 bis n^2 durchnummeriert werden. Im folgenden soll eine Version „immer an der Wand entlang“ durch schrittweise Verfeinerung umgesetzt werden. Die durchzunummerierende Matrix M ist durch eine zweistufige Reihung der Dimension $(n+2)\times(n+2)$ (inkl. Überlaufrand -1) gegeben.

Tabelle 1: Beispiel für eine Matrix mit Kantenlänge $n = 3$

-1	-1	-1	-1	-1
-1	1	2	3	-1
-1	8	9	4	-1
-1	7	6	5	-1
-1	-1	-1	-1	-1

Gehen Sie von folgendem Pseudocode aus:

```
1  //Initialisiere Matrix
2  //Setze Position auf (1, 1)
3  //Initialisiere Richtung "nach rechts"
4  for (i <- 1 to n * n) {
5    //Besetze Position mit i
6    //Bestimme mithilfe der Richtung vorlaeufige neue Position
7
8    if (vorlaeufige Position auf Rand oder bereits besetzt) {
9      //Aendere Richtung (rechts um)
10     //Bestimme mithilfe der Richtung neue Position
11   } else {
12     //Uebernimm vorlaeufige Position als neue Position
13   }
14 }
```

a) Implementieren Sie zunächst eine Funktion `init`, die zu einer gegebenen Dimension n eine zweistufige Reihung der Dimension $(n+2)\times(n+2)$ (inkl. Überlaufrand) erstellt und deren Elemente mit 0 bzw. die Elemente des Überlaufrandes mit -1 initialisiert sind.

Die Initialisierung lässt sich mithilfe von `Array.tabulate` realisieren:

```
1 def init(n : Int) = Array.tabulate(n + 2, n + 2)
2   ((r, c) => if (r == 0 || c == 0 || r == n + 1 || c == n + 1) -1
3               else 0)
```

b) Überlegen Sie sich zuerst, wie sie die Anweisungen zur Initialisierung von Position und Richtung sowie deren Änderungen umsetzen können. Welche Hilfsdatenstrukturen und/oder -funktionen sind nützlich?

Auf einfache Weise lässt sich Position und Richtung jeweils in Form eines Tupels realisieren. Die Initialisierung von Richtung und Position ergibt sich dann wie folgt:

```
1 //Setze Position auf (1, 1)
2 var pos = (1, 1)
3
4 //Initialisiere Richtung "nach rechts"
5 var dir = (0, 1)  //keine Reihe nach unten, eine Spalte nach
    rechts
```

Die notwendige Richtungsänderung sowie die Bestimmung der neuen Position ergibt sich dann elegant wie folgt:

```
1 //Aendere Richtung (rechts um)
2 dir = (dir._2, -dir._1)
3
4 //Bestimme mithilfe der Richtung neue Position
5 pos = (pos._1 + dir._1, pos._2 + dir._2)
```

Hinweis: Alternativ können Sie auch alle vier möglich Richtungen in einen Array schreiben und jeweils in einem Index festhalten, welche Richtung gegenwärtig aktuell ist. Bei einer Richtungsänderung ist lediglich der Index zu erhöhen (und bei Erreichen von 4 auf 0 zu setzen).

c) Überführen Sie nun den Pseudocode in ein lauffähiges Scala-Programm.

Die Überführung der übrigen Zeilen liefert folgenden Scala-Quellcode:

```
1  var M = init(n)                  //Initialisiere Matrix
2  var pos = (1, 1)                 //Setze Position auf (1, 1)
3  var dir = (0, 1)                 //Initialisiere Richtung "nach
       rechts"
4
5  for (i <- 1 to n * n) {
6    //Besetze Position mit i
7    M(pos._1)(pos._2) = i
8
9    //Bestimme mithilfe der Richtung vorlaeufige neue Position
10   var tmp = (pos._1 + dir._1, pos._2 + dir._2)
11
12   //vorlaeufige Position auf Rand oder bereits besetzt?
13   if (M(tmp._1)(tmp._2) != 0) {
14     //Aendere Richtung (rechts um)
15     dir = (dir._2, -dir._1)
16
17     //Bestimme mithilfe der Richtung neue Position
18     pos = (pos._1 + dir._1, pos._2 + dir._2)
19   }
20   else {
21     //Uebernehme vorlaeufige Position als neue Position
22     pos = tmp
23   }
24 }
```