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

WT 2020
Übungsblatt 4

Einführung in die Informatik II

07. und 10.02.2020

1 Schach – Turmbedrohung

In dieser Aufgabe soll ein Scala-Programm entwickelt werden, welches die Koordinaten aller von weißen Türmen bedrohten Felder auf einem Schachbrett ausgibt und zu jedem dieser Felder angibt, ob es leer ist oder welche gegnerische Figur es enthält.

Folgender Pseudocode skizziert einen möglichen Lösungsansatz:

```
1 //Bestimme die Positionen aller weissen Tuerme auf einem Schachbrett
2
3 for (Position <- Positionen aller weissen Tuerme) {
4   //Gib alle bedrohten Felder oberhalb von Position aus
5   //Gib alle bedrohten Felder rechts von Position aus
6   //Gib alle bedrohten Felder unterhalb von Position aus
7   //Gib alle bedrohten Felder links von Position aus
8 }
```

Für die Implementierung sind folgende Definitionen vorgegeben:

```
1  object ChessmanType extends Enumeration {
2    type ChessmanType = Value
3    val King, Queen, Rook, Bishop, Knight, Pawn = Value
4  }
5
6  object Color extends Enumeration {
7    type Color = Value
8    val White, Black = Value
9  }
10
11 import ChessmanType._
12 import Color._
13
14 abstract class FieldContent
15 case class Chessman(cmt : ChessmanType, color : Color) extends
       FieldContent
16 case class Empty() extends FieldContent
17
18 def printField(row : Int, col : Int, fc : FieldContent) : Unit = {
19   println("(" + row + ", " + col + "): " + fc)
20 }
21
22 abstract class Position
23 case class Coord(row : Int, col : Int) extends Position
24 case class Undef() extends Position
25
```

```
26 type Chessboard = Array[Array[FieldContent]]
27
28 def emptyChessboard : Chessboard = Array.fill(8, 8)(Empty())
```

a) Implementieren Sie die Bestimmung der Positionen aller weißen Türme (Typ: `Rook`) als Methode. Zur weiteren Verwendung sind die Positionen in geeigneter Weise in einer Variablen zu speichern.

Als Variable wählen wir zunächst `whiteRooks`, in der die Positionen der maximal 10 weißen Türme gespeichert werden sollen. (Jeder Spieler besitzt am Anfang zwei Türme, kann aber theoretisch jeden seiner acht Bauern durch einen Turm ersetzen und daher insgesamt bis zu 10 Türme besitzen). Die Anweisung zur Bestimmung aller weißen Türme lässt sich nun wie folgt verfeinern:

```
1 def getWhiteRookPositions(cb: Chessboard): Array[Position] = {
2   //Initialisiere whiteRooks
3   for (row <- 0 to 7; col <- 0 to 7)
4     //Turm eintragen, falls an Position ein weißer Turm steht
5   }
6 }
```

Für die Implementierung bietet sich nun als Variablentyp für `whiteRooks` eine zehnelementige Reihung an. Zudem benötigen wir eine Variable (`cWR`), die die Anzahl weißer Türme speichert, um festzuhalten, bei welchem Index die nächste Position gespeichert werden kann. Die Implementierung sieht dann wie folgt aus:

```
1  def getWhiteRookPositions(cb: Chessboard): Array[Position] = {
2    var whiteRooks : Array[Position] = Array.fill(10)(Undef())
3    var cWR = 0
4
5    for (row <- 0 to 7; col <- 0 to 7) {
6      if (cb(row)(col) == Chessman(Rook, White)) {
7        whiteRooks(cWR) = Coord(row, col)
8          cWR += 1
9      }
10   }
11   return whiteRooks
12 }
```

b) Implementieren Sie die Methode `nextField`, die zu einer gegebenen Position und einer angegebenen Richtung die nächstliegende Position bestimmt. Ist die Position dabei außerhalb des zulässigen Bereichs, ist `Undef()` zurückzugeben. Nutzen Sie folgende Definitionen:

```
1 type Direction = (Int, Int)
2 val LEFT = (0, -1)
3 val RIGHT = (0, 1)
4 val UP = (-1, 0)
5 val DOWN = (1, 0)
6
7 def nextField(p : Position, dir : Direction) : Position = ...
```

Die neuen Koordinaten ergeben sich aus den alten Koordinaten der Position `p` und den Komponenten der Richtung `dir`. Eine mögliche Implementierung ist daher wie folgt:

```
1 def nextField(p : Position, dir : Direction) : Position = p match
      {
2   case Undef() => return Undef()
3   case Coord(row, col) => {
```

```
4       val nr = dir._1 + row
5       val nc = dir._2 + col
6       if(nr < 0 }} nr >= 8 }} nc < 0 }} nc >= 8)
7         return Undef()
8       else
9         return Coord(nr, nc)
10    }
11  }
```

c) Implementieren Sie eine Hilfsprozedur, die zu einem übergebenem Schachbrett, einer angegebenen Position und einer gegebenen Richtung alle bedrohten Positionen und den jeweiligen Inhalt ausgibt.

Hier empfiehlt sich folgender Ansatz:

```
1  def threatFieldsInDir(cb : Chessboard, pos : Position,
2    dir : Direction) {
3    var p = nachstes Feld in Richtung dir
4
5    while (p gueltig) {
6      if (Feld an Position p ist leer) {
7        Feldinhalt ausgeben
8        p = nachstes Feld in Richtung dir
9      }
10     else {
11       if (schwarze Figur an Position p) {
12         Feldinhalt ausgeben
13       }
14       Schleife beenden
15     }
16   }
17 }
```

Eine Implementierung ist wie folgt möglich:

```
1  def printThreatenedFieldsInDir(cb : Chessboard, pos : Position,
2    dir : Direction) {
3    var p = nextField(pos, dir)
4
5    while (p != Undef()) {
6      val Coord(row, col) = p
7      val field = cb(row)(col)
8
9      if (field == Empty()) {
10       printField(row, col, field)
11       p = nextField(p, dir)
12     }
13     else {
14       val Chessman(_, clr) = field
15       if (clr == Black) printField(row, col, field)
16
17       p = Undef()
18     }
19   }
20 }
```

d) Implementieren Sie die Prozedur `printThreatenedFields`, die alle von weißen Türmen bedrohten Felder ausgibt. Nutzen Sie die Teilergebnisse aus den vorherigen Teilaufgaben.

```
1 def printThreatenedFields(cb : Chessboard) : Unit = ...
```

Die Implementierung ergibt sich schließlich aus der Verfeinerung des eingangs aufgeführten Pseudocodes unter Verwendung der vorherigen Teilergebisse:

```
1 def printThreatenedFields(cb : Chessboard) : Unit = {
2   var whiteRooks = getWhiteRookPositions(cb);
3
4   for (pos <- whiteRooks; if pos != Undef()) {
5     for (dir <- List(UP, RIGHT, DOWN, LEFT)) {
6       threatenedFieldsInDir(cb, pos, dir)
7     }
8   }
9 }
```

e) Überlegen Sie sich, wie sich der durch den Pseudocode beschriebene Ansatz erweitern lässt, sodass er für alle „Linienfiguren“ (Turm, Läufer, Dame) anwendbar ist. Welche Änderungen sind in der Implementierung notwendig?

Zunächst ist der Pseudocode wie folgt anzupassen (X ist eine Linienfigur):

```
1 Bestimme die Positionen aller weissen X auf einem Schachbrett
2
3 for (Position <- Positionen aller X) {
4   for (Richtung <- Moegliche Richtungen von X) {
5     Gib alle bedrohten Felder in Richtung
6       ausgehend von Position aus
7   }
8 }
```

Sämtliche Funktionen müssten nun so angepasst werden, dass eine Liste (oder ein Array) mit Richtungen beim Aufruf übergeben wird und diese (oder dieser) anstatt der hardcodierten Liste verwendet wird. Natürlich ist zusätzlich erforderlich, die entsprechende Linienfigur beim Aufruf von `printThreatenedFields` auch mit anzugeben, um diese auf dem Schachbrett finden zu können.

Merke: Oft erspart man sich späteren unnötigen Implementierungsaufwand, wenn man sich zu Beginn nicht auf Einzelfälle konzentriert.

2 Flaggenproblem

Eine Reihung, bestehend aus Elementen des Aufzählungstyps `Farbe` (mit den Werten `Schwarz`, `Rot` und `Gold`) soll nach den Farben in der Reihenfolge `Schwarz-Rot-Gold` aufsteigend sortiert werden. Es sind zwei Lösungsansätze denkbar, bei denen sich die Aufteilung der vier Bereiche (*schwarz (s)*, *rot (r)*, *gold (g)* und *unsortiert (u)*) unterscheiden. Zum einen können die Bereiche in der Reihenfolge „*s-r-g-u*“aufgeteilt werden, alternativ sieht der zweite Ansatz die Aufteilung „*s-r-u-g*“vor.

a) Begründen Sie, inwiefern sich die Aufteilung der Bereiche positiv auf die Anzahl notwendiger Vertauschungen auswirkt.

> In dem ersten Lösungsansatz sind die Bereiche (*schwarz*, *rot*, *gold* und der unsortierte) unmittelbar aufeinanderfolgend angeordnet. Der unsortierte Bereich wird mit jedem Schleifendurchlauf nur von links um jeweils ein Element verkleinert. Solange das jeweils nächste zu betrachtende Element e nicht *gold* ist, ist mindestens eine Vertauschung erforderlich:
>
> - Ist e *rot*, dann ist ein *rotes* Element mit einem *goldenen* zu vertauschen.
> - Ist e *schwarz*, dann ist zuächst ein *schwarzes* Element mit einem *roten* und anschließend ein *rotes* mit einem *goldenen* zu vertauschen.
>
> Bei dem zweiten Lösungsansatz sind die Bereiche so angeordnet, dass sich der unsortierte noch zu bearbeitende zwischen dem *roten* und dem *goldenen* befindet. Wird auch hier der Bereich von links nach rechts durchlaufen, ist nur dann eine Vertauschung notwendig, wenn das nächste zu betrachtende Element entweder *schwarz* oder *gold* ist. Anders als beim ersten Lösungsansatz ist aber immer nur eine Vertauschung notwendig. Stellt man zudem zu Beginn eines jeden Schleifendurchlaufs sicher, dass durch entsprechende Verringerung der rechten Indexgrenze des unsortierten Bereichs am rechten Rand keine goldenen Elemente mehr stehen, dann kann man die Anzahl nötiger Vertauschungen weiter verringern.

b) Vervollständigen Sie die Prozedur `sortiereFlagge`, die eine übergebene Flagge gemäß dem zweiten Lösungsansatz sortiert:

Hinweis: Es ist von Vorteil sich hier eine Skizze mit den verschiedenen Indizes und möglichen Zuständen des Flaggen-Arrays zu machen.

```
1  object Farbe extends Enumeration {
2    type Farbe = Value
3    val Schwarz, Rot, Gold = Value
4  }
5  import Farbe._
6
7  type Flagge = Array[Farbe]
8
9  def sortiereFlagge(flag : Flagge) : Unit = {
10   val n = flag.length
11
12   // Initialisierung von Indizes für die (sortierten) Bereiche
13   // _s_chwarz, _r_ot und _g_old
14   var s = -1 // Ende des Bereichs schwarz
15   var r = s  // Ende des Bereichs rot
16   var g = n  // Anfang des Bereichs gold
17
18   // "unsortierter" Bereich ist zwischen dem Ende von rot und Anfang
19   // von gold -> solange Vertauschen bis der Bereich leer ist
20   while (r + 1 != g){
21     //Verkleinere Indexbereich von r + 1 bis g - 1 durch
22     //Vertauschen und Aktualisieren der Indizes
```

```
23      ...
24    }
25  }
```

In jedem Schleifendurchlauf wird der unsortierte Bereich um mindestens eins verkleinert. Solange der Bereich allerdings mit roten Elementen beginnt und/oder mit goldenen Elementen endet, sind keine Vertauschungen notwendig und eine Verkleinerung kann bereits durch Indexaktualisierungen erreicht werden. In der folgenden Implementierung wird diesem Umstand Rechnung getragen:

```
1  def sortiereFlagge(flag : Flagge) : Unit = {
2    val n = flag.length
3
4    var s = -1
5    var r = s
6    var g = n
7
8    while (r + 1 != g){
9
10     //Nur Vertauschen, wenn noetig: Indexbereich soweit
         verkleinern,
11     //dass flag(r + 1) != Rot und flag(g - 1) != Gold
12     while (r + 1 < n && flag(r + 1) == Rot) r += 1
13     while (g - 1 > 0 && flag(g - 1) == Gold) g -= 1
14
15     //Pruefen, ob Abbruchbedingung noch gilt
16     if (r + 1 != g) {
17       if (flag(r + 1) == Schwarz) {
18         //Erstes rote Element mit gefundenem schwarzen
19         //Element tauschen
20         flag(r + 1) = flag(s + 1) //entspricht Rot
21         flag(s + 1) = Schwarz
22         s += 1
23         r += 1
24       } else if (flag(r + 1) == Gold) {
25         //Letztes nichtgoldene Element mit
26         //gefundenem goldenen Element tauschen
27         flag(r + 1) = flag(g - 1)
28         flag(g - 1) = Gold
29         g -= 1
30       }
31     }
32   }
33 }
```