Lab 6. For expressions

6.1 A small string DSL

In this section we will extend the scala `String` class with several operators that implement useful string functions, effectively obtaining a small string-specialized DSL.

6.1.1 Implement the `<<` and `>>` operators for a `String` and an `Int`, that shift the string in a direction by the given amount:

```
scala> "odersky" << 2
val res55: String = ersky


scala> "odersky" >> 2
val res56: String = oders
```

6.1.2 Implement the `<<<` and `>>>` operators for a `String` and an `Int`, that **rotate** the string in a direction by the given amount:

```
scala> "odersky" >>> 2
val res57: String = kyoders


scala> "odersky" <<< 2
val res58: String = erskyod
```

6.1.3 Implement the `-` operator between two strings that removes occurrences of the second string from the first one:

```
scala> "Implement the ''-'' operator between two strings that removes occurrences of
the second string from the first one:" - "th"
val res61: String = Implement e ''-'' operator between two strings at removes
occurrences of e second string from e first one:
```

6.1.4 Implement the `~` unary operator that changes the case of each cased character in the string:

```
scala> ~"xXx1337ScAlAcOdErxXx"
val res62: String = XxX1337sCaLaCoDeRXxX
```

You will have to use the following signature:

```
def unary_~ = ...
```

6.1.5 Implement the `<=>` operator that behaves like C's `strcmp` functions: if both strings are equal, returns 0; otherwise, if the first string is lexicographically first, returns a negative values and if the first string is lexicographically seond, returns a positive value:

```
scala> "haskell" <=> "java"
val res3: Int = -1


scala> "scala" <=> "java"
val res4: Int = 1


scala> "scala" <=> "scala"
val res5: Int = 0
```

This is known as "the spaceship operator".

6.1.6 Implement the `/` operator that splits a string into chunks of a given length:

```
scala> "scalable language" / 3
val res9: List[String] = List(sca, lab, "le ", lan, gua, ge)
```

6.2 Tic-Tac-Toe

Tic Tac Toe is usually played on a 3×3 board, marking positions by each player in rounds. Our game is slightly different (usually called 5-in-a-row):

- it can be played on a square board of any size **larger or equal to 5**.
- A player wins if it has marked a line, column or diagonal of **5 consecutive positions** in a row.

Example of a winning position for `X` on a 5×5 board:

```
X...0
0X.0.
..X0.
...X.
.0..X
```

Example of a winning position for `0` on a 7×7 board:

```
.X...X.
```

```
...0...
...0...
.X.0..X
0..0..0
...0...
...X...
```

Encodings

- In your project template, `X` is encoded as the **first** player (`One`), and `0`, as `Two`.

```
trait Player {}
case object One extends Player {
  override def toString: String = "X"
}
case object Two extends Player {
  override def toString: String = "0"
}
case object Empty extends Player {
  override def toString: String = "."
}
```

- A `Board` is encoded as a List of Lists of **positions** (i.e. a matrix), where a position can be `One`, `Two` or `Empty`. We make no distinction in the code between a position and a player, although `Empty` cannot be seen as a valid player. This makes the code slightly easier to write.

```
type Line = List[Player]
type BoardList = List[Line]


case class Board(b: BoardList) {
  override def toString: String = ???
}
```

Tasks

The following functions have a given signature. However, it is up to the student to decide whether these will be methods of a class or just simple functions.

6.2.1. Write a function which converts a string into a `Board`. As a helper, you can use `_.split( c )` where c is a separator string, and `_.toList`. The best solution is to use a combination of `map` calls with the above mentioned functions. A string is encoded exactly as in the examples shown above:

- there are no whitespaces - empty positions are marked by the character '.'
- lines are delimited by '\n' (the last line does not have a trailing '\n').

```
def makeBoard(s: String): Board = {
    def toPos(c: Char): Player =
      c match {
        case 'X' => One
        case '0' => Two
        case _ => Empty
      }
    ???
}
```

6.2.2. Write a function which checks if a position on the board is free. Recall that list indexing can be done using `l(_)`. Positions are numbered from 0.

```
def isFree(x:Int, y:Int):Boolean = ???
```

6.2.3. Write a function which returns the *opponent* of a player:

```
def complement(p: Player): Player = ???
```

6.2.4. We want to write a function which converts a board to a string, following the same strategy. Complete the `toString` in the Board class. Hint: instead of `foldRight`, you can use `reduce` which works quite similarly, but without requiring an accumulator.

6.2.5. Write a function which returns the *columns* of a board:

```
def getColumns: Board = ???
```

6.2.6. Implement the following two functions for extracting the first and second diagonal, as lines, from a board. Hint: use for comprehensions.

```
def getFstDiag(): Line = ???
def getSndDiag(): Line = ???
```

6.2.7. Implement the following functions for extracting diagonals above/below the first/second diagonal, as lines. It's not really necessary to make sure that at least 5 positions are available, for

now. Hint: if one function must be implemented with element-by-element iteration, the three other can be implemented using each-other, as single-line calls.

```
def getAboveFstDiag: List[Line] = ???
def getBelowFstDiag: List[Line] = ???
def getAboveSndDiag: List[Line] = ???
def getBelowSndDiag: List[Line] = ???
```

6.2.8. Write a function which checks if a player is the winner. Hint: functions `l.forall(_)` and `l.exists(_)` may be very helpful, together with patterns.

```
def winner(p: Player): Boolean = ???
```

6.2.9. Write a function which updates a position from the board, with a given player. The position need not be empty and you are not required to check this. Hint: re-use an inner aux-function together with `take` and `drop`.

```
def update(p: Player)(ln: Int, col: Int) : Board = ???
```

6.2.10. Write a function which generates all possible next-moves for any of the two players. A next-move consists in a new board, where the player-at-hand played his move.

```
def next(p: Player): List[Board] = ???
```

Testing

Use the following board configurations to test your solutions:

```
val t1 =
  """X0X0X0
    |0X0X0X
    |X0X0X0
    |.XX0..
    |X00...
    |X0X0X0""".stripMargin

val t2 =
  """......
    |......
    |......
    |.XX...
```

```
    |.0000.
    |......""".stripMargin

val t3 =
  """0X0X0.
    |000.X0
    |0.0X..
    |0..0..
    |0X..0X
    |...X..""".stripMargin
```