Main Part 4:
Implementing the Shogun Board Game (8 Marks)

"The problem with object-oriented languages is they've got all this implicit, environment that they carry around with them. You wanted a banana but what you got was a gorilla holding the banana and the entire jungle."

— Joe Armstrong (creator of the Erlang programming language)

You are asked to implement a Scala program for playing the Shogun board game.

Important

- Make sure the files you submit can be processed by just calling `scala-cli compile <<filename.scala>>` on the command line.[1] Use the template files provided and do not make any changes to arguments of functions or to any types. You are free to implement any auxiliary function you might need.
- **Do not leave any test cases running in your code because this might slow down your program!** Comment out test cases before submission, otherwise you might hit a time-out.
- Do not use any mutable data structures in your submissions! They are not needed. This means you cannot create new `Array`s or `ListBuffer`s, for example.
- Do not use `return` in your code! It has a different meaning in Scala than in Java. It changes the meaning of your program, and you should never use it.
- Do not use `var`! This declares a mutable variable. Only use `val`!
- Do not use any parallel collections! No `.par` therefore! Our testing and marking infrastructure is not set up for it.

Also note that the running time of each task will be restricted to a maximum of 30 seconds on my laptop: If you calculate a result once, try to avoid to calculate the result again.

Disclaimer

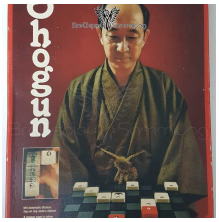
It should be understood that the work you submit represents your **own** effort! You have implemented the code entirely on your own. You have not copied

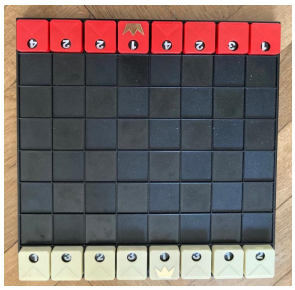
[1] All major OSes, including Windows, have a command line. So there is no good reason to not download scala-cli, install it and run it on your own computer. Just do it!

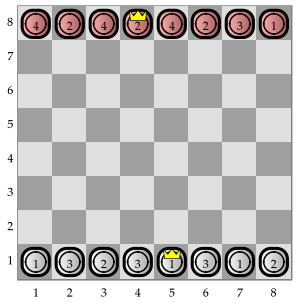
from anyone else. Do not be tempted to ask Copilot for help or do any other shenanigans like this! An exception is the Scala code I showed during the lectures or uploaded to KEATS, which you can freely use.

Background

Shogun (🔊[shōgoon]) is a game played by two players on a chess board and is somewhat similar to chess and checkers. A real Shogun board looks like in the pictures on the left.







In what follows we shall use board illustrations as shown on the right. As can be seen there are two colours in Shogun for the pieces, red and white. Each player has 8 pieces, one of which is a king (the piece with the crown) and seven are pawns. At the beginning the pieces are lined up as shown above. What sets Shogun apart from chess and checkers is that each piece has, what I call, a kind of *energy*—which for pawns is a number between 1 and 4, and for kings between 1 and 2. The energy determines how far a piece has to move. In the physical version of Shogun, the pieces and the board have magnets that can change the energy of a piece from move to move—so a piece on one field can have energy 2 and on a different field the same piece might have energy 3. There are some further constraints on legal moves, which are explained below. The point of this coursework part is to implement functions about moving pieces on the Shogun board.

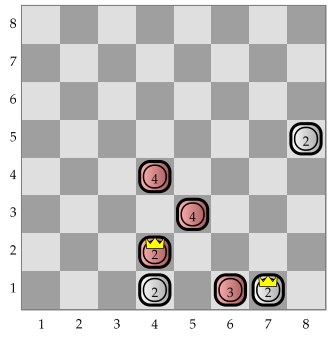
Like in chess, in Shogun the players take turns of moving and possibly capturing opposing pieces. There are the following rules on how pieces can move:

- The energy of a piece determines how far, that is how many fields, a piece has to move (remember pawns have an energy between 1 – 4, kings have an energy of only 1 – 2). The energy of a piece might change when the piece moves to new field.
- Pieces can move in straight lines (up, down, left, right), or in L-shape moves, meaning a move can make a single 90°-turn. S-shape moves with more than one turn are not allowed. Also in a single move a piece cannot go forward and then go backward—for example with energy 3 you

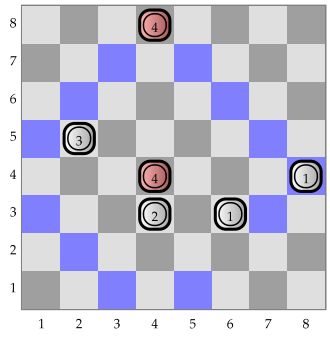
cannot move 2 fields up and then 1 field down. A piece can never move diagonally.

- A piece cannot jump over another piece and cannot stack up on top of your own pieces. But you can capture an opponent's piece if you move to an occupied field. A captured piece is removed from the board.

Like in chess, checkmate is determined when the king of a player cannot move anymore to a field that is not attacked, or cannot capture or block the attacking piece, or the king is the only piece left for a player. A non-trivial board that is checkmate is the following:



The reason for the checkmate is that the white king on field (7, 1) is attacked by the red pawn on (5, 3). There is nowhere for the white king to go, and no white pawn can be moved into the way of this red pawn and white can also not capture it. When determining a possible move, you need to be careful with pieces that might be in the way. Consider the following position:



(1)

The red piece in the centre on field (4, 4) can move to all the blue fields. In particular it can move to (2, 6), because it can move 2 fields up and 2 fields to the left—it cannot reach this field by moving two fields to the left and then two up, because jumping over the white piece at (2, 5) is not allowed. Similarly, the field at (6, 2) is unreachable for the red piece because of the two white pieces at (4, 3) and (6, 3) are in the way and no S-shape move is allowed in Shogun.

The red piece on (4, 4) cannot move to the field (4, 8) at the top, because a red piece is already there; but it can move to (8, 4) and capture the white piece there. The moral is we always have to explore all possible ways in order to determine whether a piece can be moved to a field or not: in general there might be several ways and some of them might be blocked.

Hints

Useful functions about pieces and boards are defined at the beginning of the template file. The function `.map` applies a function to each element of a list or set; `.flatMap` works like `map` followed by a `.flatten`—this is useful if a function returns a set of sets, which need to be "unioned up". Sets can be partitioned according to a predicate with the function `.partition`. For example

```
val (even, odd) = Set(1,2,3,4,5).partition(_ % 2 == 0)
// --> even = Set(2,4)
//     odd  = Set(1,3,5)
```

The function `.toList` transforms a set into a list. The function `.count` counts elements according to a predicate. For example

```
Set(1,2,3,4,5).count(_ % 2 == 0)
// --> 2
```

Tasks

You are asked to implement how pieces can move on a Shogun board. Let us first fix the basic datastructures for the implementation. A position (or field) is a pair of integers, like $(3, 2)$. The board's dimension is always 8×8. A colour is either red (`Red`) or white (`Wht`). A piece is either a pawn or a king, and has a position, a colour and an energy (an integer). In the template file there are functions `incx`, `decx`, `incy` and `decy` for incrementing and decrementing the x- and y-coordinates of positions of pieces.

A board consists of a set of pieces. We always assume that we start with a consistent board and every move only generates another consistent board. In this way we do not need to check, for example, whether pieces are stacked on top of each other or located outside the board, or have an energy outside the permitted range. There are functions `-` and `+` for removing, respectively adding, single pieces to a board. The function `occupied` takes a position and a board as arguments, and returns an `Option` of a piece when this position is occupied, otherwise `None`. The function `occupied_by` returns the colour of a potential piece on that position. The function `is_occupied` returns a boolean for whether a position is occupied or not; `print_board` is a rough function that prints out a board on the console. This function is meant for testing purposes.

(1) You need to calculate all possible moves for a piece on a Shogun board. In order to make sure no piece moves forwards and backwards at the

same time, and also exclude all S-shape moves, the data-structure `Move` is introduced. A `Move` encodes all simple moves (up, down, left, right) and L-shape moves (first right, then up and so on). This is defined as follows:

```
abstract class Move
case object U extends Move    // up
case object D extends Move    // down
case object R extends Move    // right
case object L extends Move    // left
case object RU extends Move   // ...
case object LU extends Move
case object RD extends Move
case object LD extends Move
case object UR extends Move
case object UL extends Move
case object DR extends Move
case object DL extends Move
```

You need to implement an `eval` function that takes a piece `pc`, a move `m`, an energy `en` and a board `b` as arguments. The idea is to recursively calculate all fields that can be reached by the move `m` (there might be more than one). The energy acts as a counter and decreases in each recursive call until 0 is reached (the final field). The function `eval` for a piece `pc` should behave as follows:

- If the position of a piece is outside the board, then no field can be reached (represented by the empty set `Set()`).
- If the energy is 0 and the position of the piece is *not* occupied, then the field can be reached and the set `Set(pc)` is returned whereby `pc` is the piece given as argument.
- If the energy is 0 and the position of the piece *is* occupied, but occupied by a piece of the opposite colour, then also the set `Set(pc)` is returned. Otherwise the empty set `Set()` is returned.
- In case the energy is > 0 and the position of the piece `pc` is occupied, then this move is blocked and the set `Set()` is returned.
- In all other cases we have to analyse the move `m`. First, the simple moves (that is `U`, `D`, `L` and `R`) we only have to increment / decrement the x- or y-position of the piece, decrease the energy and call eval recursively with the updated arguments. For example for `U` (up) you need to increase the y-coordinate:

  `U` $\Rightarrow$ new arguments: `incy(pc)`, `U`, energy - 1, same board

  The move `U` here acts like a "mode", meaning if you move up, you can only move up; the mode never changes for simple moves. Similarly for the other simple moves: if you move right, you can only move right and so on. In this way it is prevented to go first to the

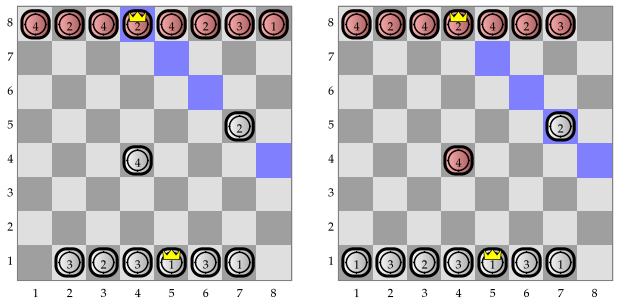
right, and then change direction in order to go left (same with up and down).

For the L-shape moves (`RU`, `LU`, `RD` and so on) you need to calculate two sets of reachable fields. Say we analyse `RU`, then we first have to calculate all fields reachable by moving to the right; then we have to calculate all moves by changing the mode to `U`. That means there are two recursive calls to `eval`:

`RU` ⇒ new args for call 1: `incx(pc)`, `RU`, energy - 1, same board
new args for call 2: `pc`, `U`, same energy, same board

In each case we receive some new piece(s) on reachable fields and therefore we return the set containing all these fields. Similarly in the other cases.

For example in the left board below, `eval` is called with the white piece in the centre and the move `RU` generates then a set of new pieces corresponding to the blue fields. The difference with the right board is that `eval` is called with a red piece and therefore the field (4, 8) is not reachable anymore because it is already occupied by another red piece. But (7, 5) becomes reachable because it is occupied by a piece of the opposite colour.

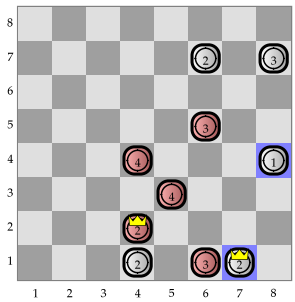
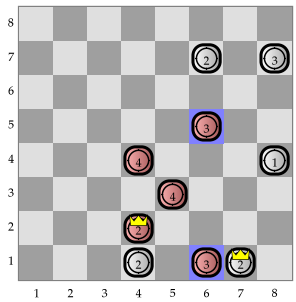


[3 Marks]

(2) Implement an `all_moves` function that calculates for a piece and a board, *all* possible onward positions. For this you have to call `eval` for all possible moves `m` (that is `U`, `D`, ..., `DL`). An example for all moves for the red piece on (4, 4) is shown in (1) on page 3. Be careful about possible modifications you need to apply to the board before you call the `eval` function. Also for this task, ignore the fact that a king cannot move onto an attacked field.

[1 Mark]

(3) Implement a function `attacked` that takes a colour and a board and calculates all pieces of the opposite side that are attacked. For example below in the left board are all the attacked pieces by red, and on the right all for white:

[1 Mark]

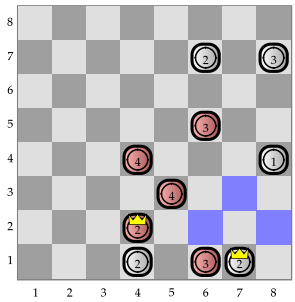
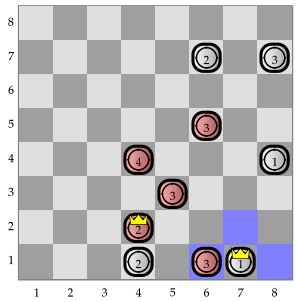
(4) Implement a function `attackedN` that takes a piece and a board and calculates the number of times this pieces is attacked by pieces of the opposite colour. For example the piece on field (8, 4) above is attacked by 3 red pieces, and the piece on (6, 1) by 1 white piece. In this number also include kings even if they cannot move to this field because the would be in "check".

[1 Mark]

(5) Implement a function `protectedN` that takes a piece and a board and calculates the number of times this pieces is protected by pieces of the same colour. For example the piece on field (8, 4) above is protected by 1 white pieces (the one on (8, 7)), and the piece on (5, 3) is protected by three red pieces ((6, 1), (4, 2), and (6, 5)). Similarly to `attackedN`, include in the calculated number here also the king provided it can reach the given piece.

[1 Mark]

(6) Implement a function `legal_moves` that behaves like `all_moves` from (2) for pawns, but for kings, in addition, makes sure that they do not move to an attacked field. For example in the board below on the left, there are three possible fields the white king can reach, but all of them are attacked by red pieces. In the board on the right where the white king has an energy of 1, there is only one legal move, namely to move to field (8, 1). The field (7, 2) is reachable, but is attacked; similarly capturing the red piece on field (6, 1) is not possible because it is protected by at least another red piece.

[1 Mark]