Assignment 1

Chess Visualisation

Specification Version: 1.1 *(Released 7th August, 2025, updated 15th August 2025)*

Changelog:

- 1.1 - bumped deadline to avoid confusion about timing between Ed and the spec.
- 1.1 - added links to our course style guide.

The deadline for Assignment 1 is **8th September, 2025 10:55pm (Canberra time, sharp).**

However, the first parts of this assignment can be completed earlier! Start early and avoid crunch-time stress!

Note: Late submissions **will not be marked** unless you have an **approved extension**. Extensions can only be applied for through the Extension App. Students with Education Access Plans still need to use this App.

Use the "Submit" button on the top-right corner to submit your code to our server. You must submit at least once before the deadline. You can submit as many times as your want before the deadline; only your latest submission counts.

Usual academic misconduct standards apply to Assignment 1 (except for Part 0, below). You must work on this assignment yourself. Plagiarism detection tools are highly advanced nowadays, **do not take the risk**.

The `ChessGraphics` Library

This assignment makes use of a *library*, a pre-written module containing types and functions that you should use to implement your solution. The `ChessGraphics` library includes a variety of built-in functions to manipulate `Picture`s, which you will use to visualise chess boards. The source code to the library has been included in your Ed Lessons scaffold, but you should not have to read the source code to understand how to use the library. Instead, we recommend you consult the automatically-generated documentation for the library. This documentation lists all the functions and types that this library makes available to you, including a brief explanation of each function.

Most importantly, this library defines a command `render` that, given a `Picture` as input, saves an image (in vector format) into a file called `svgOut.svg`. To try this out, first load the assignment in `ghci` from the Terminal:

```
ghci -Wall Chess.hs
```

Then, we'll `render` a small example `Picture`:

```
ghci> render (above queen (beside knight king))
```

To view this image, it's easiest to use the provided `showPic.html` file. To do this in Ed Lessons, *right click* the file `showPic.html` and click "Open in Web Preview". This will display your image in a basic web page from within Ed.

The "Console" that shows in the web preview can be closed. It is not used for this assignment.

The provided example picture above should look like this:

If you subsequently render a different `Picture`, you will need to reload the preview page to see the updated image. In Ed Lessons, this is achieved by pressing the reload button in the top-right of the Web Preview tab.

Part 0 (2 marks)

This part (and only this part) of the assignment **may be freely discussed** with your peers and with your tutor. No anti-plagiarism restrictions apply to Part 0.

Required Knowledge: Weeks 1-2 lecture content.
Marking: 1 mark for `blackSquare`, 1 mark for `backRankPieces`

The `ChessGraphics` library defines a `Picture` for a "white" square of a chess board, called `whiteSquare`, but it doesn't include a `Picture` for the "black" squares. I use "quotes" here because while these squares are *called* "white" and "black", typical chess boards use different colours — it suffices that one colour is the *inverse* of the other. Using the functions available to you from `ChessGraphics`, **define `blackSquare`** in the file `Chess.hs`. If you implement it correctly, then `render (beside blackSquare (beside whiteSquare blackSquare))` should produce:

Next, **define `backRankPieces`**, which is a picture consisting of all the pieces that start in the "back rank" of a Chess board, in order. When `render`ed, it should look like this:

The *Test* button in the lower-right corner can be used to conveniently test all your code against all of the examples presented in this specification. However, **these examples are not sufficient to ensure your solutions to all of the problems are correct**. Please also do your own testing! When we mark your assignment, we will run a **much larger** set of test cases on your solution.

Part 1 (13 marks)

Warning: From this part onwards, your assignment code **may not be discussed freely** and **all work must be your own**. Plagiarism checking will be performed.

Required Knowledge: Weeks 1-2 lecture content.
Marking: 10 marks for correct implementation of `piecePicture`. 3 marks for style.

Note that all parts (except for part 0) of this assignment reserve some marks for **style**. We have a style guide which gives precise instructions. In general, please consider the following style guidelines:
- Avoid repetition (while maintaining clarity).
- Choose reasonable names for variables and functions.
- Use `ghci` with `-Wall` and fix all warnings.
- Any unclear part of the program should be explained with a comment.

In `Chess.hs` we defined for you a data type called `Piece`, a product of `Colour` and `PieceType`:

```
data PieceType = King | Queen | Knight | Bishop | Rook | Pawn
    deriving (Eq, Show)
data Colour = White | Black
    deriving (Eq, Show)
data Piece = Piece Colour PieceType
    deriving (Eq, Show)
```

Your next task is to **define the function `piecePicture`**:

```
piecePicture :: Piece -> Picture
```

which, given a `Piece`, produces the appropriate `Picture` for the given `PieceType`. If the `Colour` of the `Piece` is `Black`, then the `Picture` should be `invert`ed.

If you implement it correctly, `render (beside (piecePicture (Piece White Knight)) (flipV (piecePicture (Piece Black Knight))))` should produce:

Try your own tests and examples!

Part 2 (40 marks)

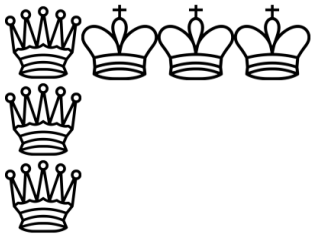
Required Knowledge: Weeks 1-3 lecture content.
Marking: For each of `repeatH`, `repeatV`: 10 marks for correct implementation, 3 marks for style. For `board`, 10 marks for correct implementation, 4 marks for style.

Implement the two functions `repeatH` and `repeatV`. These functions

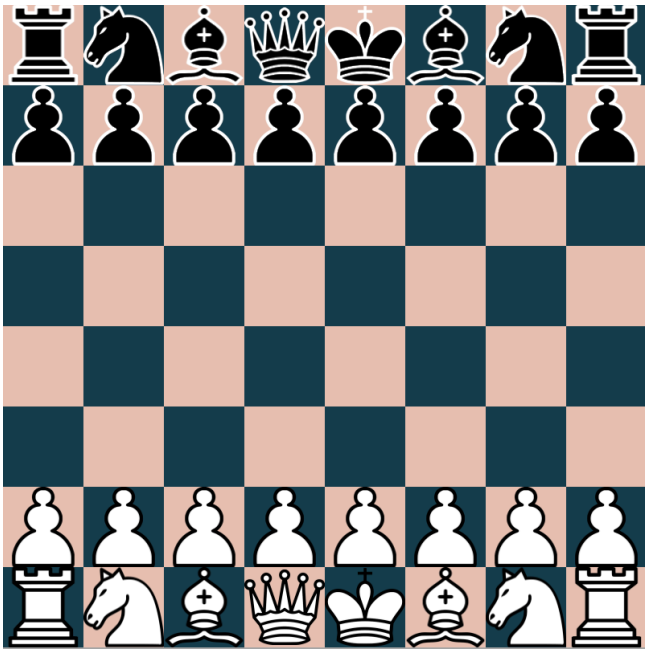
```
repeatH :: Picture -> Int -> Picture
repeatV :: Picture -> Int -> Picture
```

These two functions both take a `Picture` `p` and an `Int` `n`, and produce a `Picture` that consists of `n` copies of `p`, either stacked `above` each other (for `repeatV`) or `beside` each other (for `repeatH`). If `n <= 0`, the `Picture` produced should just be `empty`.

If implemented correctly, `render (beside (repeatV queen 3) (repeatH king 3))` should produce the following:



Next, **define `board`**, which is a `Picture` representing the Chess board in its initial configuration at the start of play. When correctly implemented, `render board` should produce:



Hint: It is often helpful to break down problems like this into subproblems. Consider dividing the board up into three parts — the top two ranks, the empty middle part of the board, and the bottom two ranks — and drawing each of these separately. These problems can be further subdivided: The top two ranks consists of the `backRankPieces` and a row of `pawn`s, placed `over` the checkerboard background, for example.

Part 3 (45 Marks)

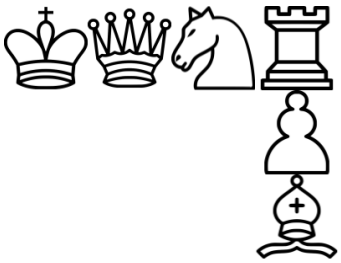
Required Knowledge: Weeks 1-4 lecture content.
Marking: For each of `combineH`, `combineV`: 10 marks for correct implementation, 3 marks for style. For `boardFor`, 15 marks for correct implementation, 4 marks for style.

Implement the two functions `combineH` and `combineV`. These functions

```
combineH :: [Picture] -> Picture
combineV :: [Picture] -> Picture
```

These two functions both take a list of `Picture`s `ps`, and produce a `Picture` that consists of each element of `ps` in order,, either stacked `above` each other (for `combineV`) or `beside` each other (for `combineH`). If `ps` is the empty list `[]`, the `Picture` produced should just be `empty`.

If implemented correctly, `render (beside (combineH [king,queen,knight]) (combineV [rook,pawn,bishop]))` should produce the following:



Lastly, the file `Chess.hs` defines two type synonyms:

```
type Position = (Int, Int) -- row,column
type GameState = [(Position,Piece)]
```

The type `Position` therefore refers to a pair of two `Int`s, *row* and *column* respectively, referring to a position on the chess board. The state of a Chess game (`GameState`) is described by a list of every `Piece` in play, along with its `Position` on the board.

> ⚠ In some chess notations, row 1 refers to the bottom of the board, while row 8 refers to the top. This is **not** the case for our `Position` type. In our type, row 0 refers to the top of the board and row 7 refers to the bottom.

Implement the function `boardFor`, which, given a `GameState` as described above, produces the `Picture` visualising that state of the chess board. You may implement this function in any way you see fit, however we offer this suggested strategy:

1. Implement a function `backdropFor :: Position -> Picture` that returns either `whiteSquare` or `blackSquare` depending on the colour of the input `Position` on the chess board.
2. Implement a function `piecePictureFor :: GameState -> Position -> Maybe Picture` that searches through the provided list (`GameState`) for a `Piece` at the provided `Position`, returning `Just` its corresponding `Picture` (using your previously-implemented

`piecePicture`) if it exists. The `lookup` function from the Prelude is helpful here.

3. The implementation of `boardFor` should use `combineV` to merge a list of all the rows of the board.
4. Each row of the board should be defined with `combineH` to merge a list of all the cells in the row.
5. Each cell should be defined as the `piecePictureFor` that position (if there is a piece there), `over` the `backdropFor` that position.

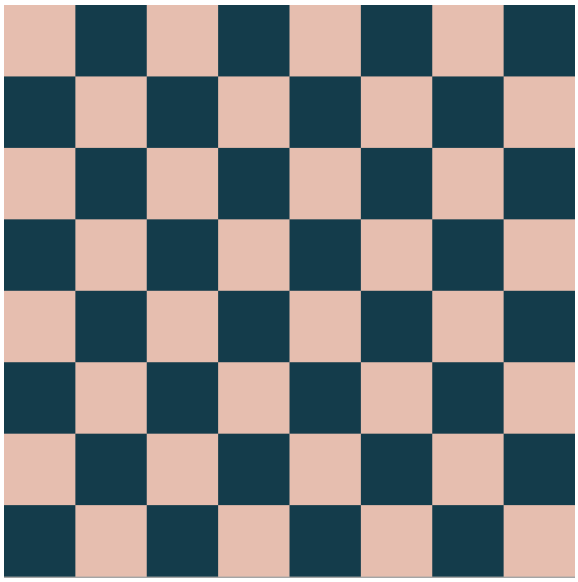
Dealing with invalid inputs

You may assume that all `Position` values only contain coordinates between `0` and `7` inclusive, and that two `Piece`s never occupy the same `Position`. Nonetheless, we encourage you to handle invalid inputs gracefully. Pieces placed at positions outside the board or which are already occupied can, in this case, be safely ignored and not displayed anywhere.

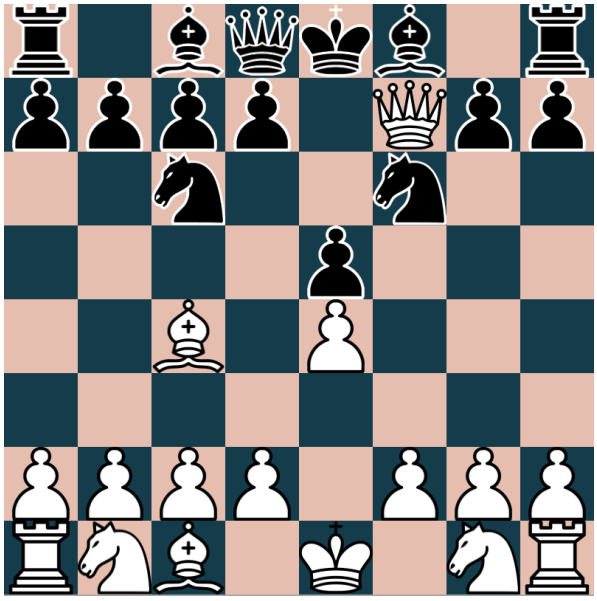
You may **not** assume any other Chess rules (e.g. that there are exactly two kings).

The file `Examples.hs` contains a few examples of `GameState` definitions you can use for your testing. Load the file by typing in the terminal `ghci -Wall Examples.hs`.

- `render (boardFor emptyBoard)` will produce an empty board as follows:



- `render (boardFor starting)` will produce the starting configuration of a game of chess, as seen above when you implemented `board`.
- `render (boardFor scholarsMate)` will produce a visualisation of the popular *Scholar's Mate*, an opening that can checkmate in 4 moves:



Please also define your own examples and test cases!

Citing Sources

If you need to cite any sources that helped with the completion of your assignment, do so in a **clearly visible comment** in `Chess.hs`. In particular, use these instructions if you need to acknowledge the use of generative AI.

⚠ You **must cite** any work that is not your own. You can only and will only be marked on work that you produced yourself.

Communicating

Do not post your code publicly, even for Task 0, on Ed Discussions or via any other means. Posts on Ed Discussions can trigger emails to students, depending on how they have configured their notifications, so if by mistake you post your code publicly, others will have access to your code and you may be held responsible for plagiarism.

Once again, and we cannot stress this enough: **do not post your code publicly** . If you need help with your code, post it *privately* to the instructors.

When brainstorming with your friends, **do not view each others' code** for anything other than Task 0. There might be pressure from your friends, but this is for both your and their benefit. Anything that hints at plagiarism will be investigated and there may be serious consequences.

Sharing concepts and sketches is perfectly fine, but sharing should stop before you risk handing in suspiciously similar solutions.

The use of generative AI tools (e.g. ChatGPT) is permitted in this course, with appropriate citation.

Course staff will not look at assignment code unless it is posted **privately** in Ed Discussions, or

shared in a drop-in consultation.

Course staff will typically give assignment assistance by asking questions, directing you to relevant exercises from the labs, or definitions and examples from the lectures. They will not give direct instructions on how to edit your Task 1, 2 or 3 code, but they will give you more generous help for questions related to Task 0.

Submission Checklist

- You have fully read and understand the entire assignment specification.
- You have fully cited any and all work (including generative AI) that you have used in **a clearly visible comment** in `Chess.hs`
- Your program compiles and runs. **The only files in the scaffold that are allowed to be changed are `Chess.hs` and `Examples.hs`. If any other files are changed, auto-testing may fail.**
- Your program runs in the Ed Lessons environment - if the program does not work in Ed Lessons, it might fail tests used by the instructors.