Black and White

Jan Rochel <jan@rochel.info>

to be handed in before 22-10

As in the previous exercise, you are asked to implement a game, this time it is Chess, or rather a framework for a class of games that can be played on a chess board, like for instance Draughts.

1 (The Pieces)**.** Define a data-type to represent the six different kind of chess pieces. Make the them an instance of *Show* such that a piece is dislayed as a single character (one of `"KQNRBP"`).

2 (The Board)**.** A chess board consits of 8x8 squares. Define a data-type (or newtype) *Board* encapsulating an *Array* (from `Data.Array`) which uses indexes of a data-type *Pos* (which you are to define). Each of the squares can hold a piece belonging either to the black or the white player. The type of piece should not be fixed since different games use different kind of pieces.

Make it an instance of *Show* such that a chess board would be printed as follows (with the white player's pieces in lower-case letters):

```
    a b c d e f g h
  +-----------------+
1 | r   b q k b n r | 1
2 | p p p p p p p p | 2
3 |     n           | 3
4 |                 | 4
5 |           P     | 5
6 |                 | 6
7 | P P P P P   P P | 7
8 | R N B Q K B N R | 8
  +-----------------+
    a b c d e f g h
```

Extend the *Board* type to a type *Situation* which also comprises which one is the next player. Make it also an instance of *Show* and *Parse*, adding a line to the above at its top which is either `"Next player: white"` or `"Next player: black"`. *Pos* should be *show*ed as e.g. `c6`.

3 (Parsing)**.** Let us define a type class *Parse* defining a function *parse* for reading in data-types from strings, with the following restriction. An instance must be an instance of *Show* and it must hold that $parse\ (show\ b) \equiv b$ and $show\ (parse\ s) \equiv s$. Make the above types an instance of this class. You can test your function on the supplied `.chb`-files.

4 (The Rules)**.** The interface of the game should work for different types of games played on a chess-board. Define a data-type *Rule* to host three functions:

- *hasLost* to check whether in a given situation the player with the next draw has lost the game
- *movesFrom* lists all possible moves for the piece at a given position
- *performMove* to move the piece from one position to another, possibly capturing pieces of the other player.

Implement a constant *chessRules* of that type with the chess rules. Note that you are only asked to implement a restricted set of rules, i.e. without castling, en passant, promotion, repetition check, etc.

5 (Invocation and Game Loop)**.** Write a main function that reads in an `.chb`-file and alternatingly asks the black and the white player to make a move. The move should be entered as `a7c8` in order to move the piece at `a7` to `c8`. Make sure to handle the cases correctly where the player doesn't have a piece at `a7` or cannot move the piece to `c8`. In the latter case you should print a list of possible target positions. After each move the board is re-displayed. The game ends if the active player cannot make a move. It is a draw unless *hasLost* holds.

6 (Look-ahead)**.** The final exercise is about implementing a look-ahead that detects wether the active player is in a winning position, i.e. wether he can win the game in the next n turns regardless of the other player's moves. The game loop is to be extended by performing this look-ahead every turn in order to print something along the lines of "Black wins in 3 turns" at the adequate situation. To this end you need to define a data-type *MoveTree* that represents all possible moves in the future. It should hold a *Situation* along with all its follow-up situations. On that tree you need to test whether there is a move for, say black, such that for all moves white can make, there again is a move for black, such that ... and so forth... white loses. In other words, you need to implement an alternation of $\forall$ and $\exists$ quantors. Make sure that the tree is not recomputed after each move.

7 (Bonus-Exercises)**.** As before you can any additional feature you have implemented as a bonus. A few suggestions here would be:

- share as much of the computation in the look-ahead between the moves as possible.

- write the rules to a different chess-board game such as Draughts or Antichess.
- produce graphical output for instance via SVG as in the previous assignment
- undo the last move

Remark: In this assignment signatures are not supplied and there are less guidelines on how to implement the functions. You are thus given more freedom in this exercise. Use it wisely. If necessary introduce not only types asked for in this assignment, make use of the module system, introduce abstractions in order to avoid code duplication.

Important: Only code that passes the type checker is accepted. If you need to leave gaps in your code use *undefined*.