ECS 140B
Spring Quarter 2018
Team Project #2

Due: no later than 6:00pm, Monday, May 21, 2018 via Canvas

Capture the Flag (CTF) is played on a square board with N spaces to a side. The number of pieces that each player may start with is variable, but each player will start with one flag and at least one pawn. Here's an example of an initial CTF board where N = 5:

```
-   w   W   w   -

-   w   w   w   -

-   -   -   -   -

-   b   b   b   -

-   b   B   b   -
```

In Capture, the two opponents are "white" and "black". Each side starts with one flag (indicated by W for the white flag and B for the black flag) and one or more pawns (indicated by w or b). Players alternate moves and try to win by achieving one of four different goal states:

1) A player wins if all the opponent's pawns have been captured
2) A player wins if the opponent's flag has been captured
3) A player wins if it's the opponent's turn to move and the opponent can't make a legal move
4) A player wins if the player moves his flag forward past all the opponent's pawns without the flag being captured.

How does a piece move? How does it capture another piece? It depends on what kind of piece it is.

Pawns can move one space forward (to the opponent's end of the board), to the left, or to the right, so long as the pawn moves to an unoccupied space. Pawns can never move backward (toward its own end of the board), and they can never move diagonally.

Pawns can jump two spaces forward, to the left, or to the right, so long as the pawn moves to an unoccupied space AND the pawn jumps over an opponent's piece. The opponent's piece is then captured and removed from the board. Pawns can never jump backward, and they can never jump diagonally. Multiple jumps are not permitted.

A flag can move one space forward, to the left, to the right, or backward. A flag can never move diagonally, and it can never move more than one space on a turn. Thus, a flag can neither jump nor capture.

Oh, there's one more constraint on movement. No piece may make a move that results in a board configuration that has occurred previously in the game. This constraint prevents the "infinite cha-cha" where one player moves left, the other player moves left, the first player moves back, the other player moves back, the first player moves left, and so on. It will be easy for you to prevent this sort of annoying behavior by checking the history list of moves that will be passed to your program.

Your team's task is to construct a Haskell function, along with all the necessary supporting functions, which takes as input a representation of the state of a CTF game (i.e., current board position plus history) and an indication as to which player is to move next. (As you read on, you'll find that the current board position is actually the first element on a list containing all the boards or states that the game has passed through, from the initial board to the most recent board.) Your top-level function should be called 'capture'.

This function returns the best next move that the function can make from that given board position. The result should be represented as a CTF board position in the same format that was used for the original board position that was passed to your function.

Your function must select the best next move by using MiniMax search. You will need to devise a static board evaluation function to embody the strategy you want your 'capture' program to employ, and you'll need to construct the necessary move generation capability.

Here's an example of how your 'capture' function will be called. Assume that we want your function to make the very first move in a game of CTF, and that N is set to 5. As we noted above, that beginning board might look like this:

```
        -   w   W   w   -

        -   w   w   w   -

        -   -   -   -   -

        -   b   b   b   -

        -   b   B   b   -
```

Your 'capture' function must then be ready to accept exactly three parameters when called. The sample function call explains what goes where in the argument list:

```
capture    ["-wWw--www-------bbb--bBb-"]     'w' 2

                ^                             ^   ^
                |                             |   |
  The first argument is a list of lists.      |   |
  That list represents a history of the       |   |
  game, board by board.  The first sublist    |   |
  on this list will be the most recent board. |   |
  The last element of the list will be the    |   |
  initial board before either player has      |   |
  moved.  This history list is initialized    |   |
  as shown above.  Each sublist is a list     |   |
  of symbols which can be either w, W, b,     |   |
  B, or -.                                    |   |
                                              |   |
  Each of these elements represents a         |   |
  space on the board.  The first N elements   |   |
  are the first or "top" row (left to         |   |
  right), the next N elements are the         |   |
  second row, and so on.                      |   |
```

The second argument will always be 'w' or 'b', to indicate whether your function is playing the side of the white pieces or the side of the black pieces. There will never be any other color used.

The third argument is an integer to indicate how many moves ahead your minimax search is to look ahead. Your function had better not look any further than that.

This function should then return the next best move, according to your search function and static board evaluator. So, in this case, the function might return:

"-wWw--w-w---w---bbb--bBb-"

or some other board in this same format. That result corresponds to the following diagram:

```
-   w   W   w   -

-   w   -   w   -

-   -   w   -   -

-   b   b   b   -

-   b   B   b   -
```

One other thing: Your 'capture' program can't assume anything about the initial board layout other than that the board will be square, each side will start with only one flag, and each side will start with at least one pawn. So there's a whole lot of possible starting boards for any given N. Your program will just have to deal with it.

Some extra notes:

1) We may need to modify the specifications a bit here or there in case we forgot something. Try to be flexible. Don't whine. Remember, this game is invented by someone who is lousy at board games. It may be awful. Do the best you can with it. You and your code should be, as we said, flexible.

2) A static board evaluation function is exactly that -- static. It doesn't search ahead. Ever.

3) You can convert our board representation to anything you want, just as long as when we talk to your top-level function or it talks to us, that function is using our representation. This is not intended as an endorsement of the board representation we've chosen here; you could easily come up with a more convenient representation scheme. We're only using this representation to give you the opportunity to think about your knowledge representation scheme.

4) We are not asking you to write the human-computer interface so that your program can play against a human. You can if you want to, just

for fun, but we don't want you to turn that in.

5) At the beginning of a game, the white pieces will be at the top of the board, and the black pieces will be at the bottom. White always moves first. But your program must be able to handle any arrangement of pieces on the board at any point in the game, and it must be able to play either the white side or the black side.

6) Program early and often.

7) Before writing any code, play the game a few times.

8) The simple commenting for every function you write should include a brief description of what the function does, the arguments that the function expects, and the value that the function returns.

9) In addition, you must include an English-languagae description of the strategies (or are they tactics?) embodied in your static board evaluation function with your submission.