CPSC 312 - Fall 2014
Project 1 - due Friday, November 7, 2014 at 6:00pm

Crusher is played on a hexagonal board with N hexes to a side. Each player starts with 2N-1 pieces arranged in two rows at opposite ends of the board. Here's an example of an initial Crusher board where N = 3:

```
      W   W   W

    -   W   W   -

  -   -   -   -   -

    -   B   B   -

      B   B   B
```

White always begins at the top of the board, and white always makes the first move.

In Crusher, players alternate moves and try to win by obliterating the other player's pieces. A piece can move in one of two ways. First, a piece can slide to any one of six adjacent spaces so long as the adjacent space is empty. So in the diagram below, the black piece can slide to any of the spaces indicated by a "*":

```
      W   W   W

    -   *   *   -

  -   *   B   *   -

    -   *   *   -

      B   B   B
```

The other type of movement is a leap. A piece can leap over an adjacent piece of the same color in any of six directions. The space that the piece leaps to may be empty, or it may be occupied by an opponent's piece. If the space is occupied by an opponent's piece, that piece is removed from the game. Thus leaping is not only a means of movement, but it's the only means of capturing an opponent's piece. Also, note that a player must line up two pieces in order to capture an opponent's piece. Here's an example of leaping. Let's say the board looks like this:

```
      W   W   -

    -   W   -   -

  -   -   B   -   -

    B   -   B   -

      -   -   -
```

If it's now black's turn, black has two possible leaps available (in addition to several slides). Black could leap like this and crush the white piece (hence the name Crusher...we we're going to use Squash, but that's already been taken):

```
      W   W   -

    -   B   -   -

  -   -   B   -   -

    B   -   -   -

      -   -   -
```

This would seem to be a pretty good move for black, as it results in a win for black. The other possible leap shows black running away for no obvious reason:

```
      W   W   -

    -   W   -   -

  -   -   -   -   -

    B   -   B   -

      -   -   B
```

Note that a piece may not leap over more than one piece. Oh, there's one more constraint on movement. No player 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 forward, the other player moves forward, the first player moves back, the other player moves back, the first player moves forward, 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.

How does a game end? Who wins? One player wins when he or she has removed N (i.e., more than half) of the opponent's pieces from the board. That was easy, wasn't it? **A player also wins if it's the opponent's turn and the opponent can't make a legal move.**

Over the next three weeks (minus a couple of days), you are to construct a Haskell function, along with all the necessary supporting functions, which takes as input a representation of the state of a Crusher game (i.e., a board position), an indication as to which player is to move next, and an integer representing the number of moves to look ahead. (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 function, which you will call "crusher", will also be passed an integer representing N (the number of hexes or spaces along one side of the board).

This function determines the best next move that the designated player can make from that given board position. That move should be represented as a Crusher board position in the same format that was used for the input board position. That new board position is then cons'ed onto the history list of boards that was passed as an argument to your function, and that updated list is the value returned by the 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 Crusher program to employ, and you'll need to construct the necessary move generation capability.

Here's an example of how your Crusher function will be called. Assume that we want your function to make the very first move in a game of Crusher, and that N is set to 3. As we noted above, that beginning board would look like this:

```
          W   W   W

        -   W   W   -

      -   -   -   -   -

        -   B   B   -

          B   B   B
```

Your Crusher function must then be ready to accept exactly four parameters when called. The sample function call explains what goes where in the argument list (the _r2d2 part will be explained below):

```
*Main> crusher_r2d2 ["WWW-WW-------BB-BBB"] 'W' 2  3
                     ^                      ^   ^   ^
                     |                      |   |   |
 The first argument is a list of strings.   |   |   |
 That list represents a history of the      |   |   |
 game, board by board.  The first string    |   |   |
 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 characters which can be either 'W', '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+1 elements are the      |   |   |
 second row, and so on.  (The number of     |   |   |
 spaces or hexes in each row increases      |   |   |
 by 1 to a maximum of 2n-1 and then         |   |   |
 decreases by 1 in each of the following    |   |   |
 rows until the bottom row, which contains  |   |   |
 n hexes or spaces.                         |   |   |
                                            |   |   |
                                            |   |   |
                                            |   |   |
```

The second argument is always '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 colour 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.

The fourth argument is an integer representing N, the dimensions of the board. The value 3 passed here says that this board has 3 spaces or hexes along each of its six sides.

This function should then return the next best move, according to your search function and static board evaluator, cons'ed to the front of the list of game boards that was originally passed to your function as the first argumment. So, in this case, the function might return:

```
["-WW-WW---W---BB-BBB", "WWW-WW-------BB-BBB"]
```

(or some other board in this same format). The new first element of this history list represents the game board immediately after the function moves a piece. That game board corresponds to the following diagram:

```
    -   W   W
  -   W   W   -
-   -   W   -   -
  -   B   B   -
    B   B   B
```

The deliverables for this project are due Friday, November 7, no later than 6:00pm. When you submit your program, make sure you provide sufficient documentation explaining where the three main operations (i.e., move generation, board evaluation, minimax) are being carried out. If your TAs can't locate these components, your marks for this project are likely to be minimal.

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, we invented this game. It may be awful. Do the best you can with it.

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 function or it talks to us, your function communicates with us using our representation.

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 to see it as part of what you submit for evaluation.

5) Program early and often. The board evaluator is easy. The rest is much more difficult. Get the board evaluator out of the way in a hurry, then start working on the rest of it as soon as you can. Get everything else working, then go back and tune your evaluator.

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

7) You'll be getting at least one and maybe two small Prolog programming assignments during the time you're working on this project. Manage your time well.

8) We may load your functions and some other team's functions into the same space, and there could be function name conflicts. To prevent these conflicts, we ask you to create unique names for every function you create. Here's how we want you to do this: Choose one of your team member's CS undergraduate email accounts — the ones that look like x1y2 — and add that four-character unique identifier, preceded by an underscore, to the end of every one of your function names, just like we did with the crusher_r2d2 example above. Your top level function should be named "crusher_x1y2", where "x1y2" is the four-character email account name you've decided to use. Also, your top-level function should be the first function defined in the file you send us. Don't make us hunt for it.

9) As we have stated many times in class, this is a two-person team project. No one-person efforts will be evaluated. For our purposes, "team project" means that both students invest equivalent efforts in the development of a single program that will be submitted for evaluation. The two team members may not work individually on two different programs and then submit only the work of one team member. The two team members also may not choose to let one student do most or all of the development while the other provides moral support.

10) For handin purposes, this is 'project1'.