Project2: Crusher (Game Playing)

23 November, 2015

Due on the midnight of Saturday, December 5th, 2015.

Collaboration Policy:

Work in the same groups as your first project, unless you have an internal deal with your teammates to work differently. The only exception is the group of four: Please split into two groups of two.

Submission:

Every team should submit one .hs (or .lhs) file where all their code is. Provide extensive comments inside the file, explaining what each function does, how it works and how it fits into the overall solution. Code with no documentation will not be marked. Also make sure in your documentation you explain where the main operations (e.g., move generation, board evaluation, minimax, search) are being carried out. If your TAs can't locate these components, your marks for this project are likely to be minimal.

The handin name for this submission will be **project2** with the course name **cs312**. Handin will be open for submission starting a week before the deadline, Nov 28th, until the midnight of Dec 8th, A penalty of 20% will apply to late submissions.

Please include the official full name, student id and ugrad id of every person in the group in your submission. Each team should make only one submission.

This project is developed by Kurt Eiselt.

The project description starts on the next page.

Part 1: The Game

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):

```
    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.

There are two ways for a player to win this game (the only ways for the game to end):
1. a player wins when he or she has removed N (i.e., more than half) of the opponent's pieces from the board.
2. A player wins if it's the opponent's turn and the opponent can't make a legal move.

Part 2: Your Task
You are to construct a Haskell function called "crusher" (along with all the necessary supporting functions) which takes as input 1) a representation of the state of a Crusher game (i.e., a board position), 2) an indication as to which player is to move next, and 3) 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.) 4) an integer representing N (the number of hexes or spaces along one side of the board).

This function determines the next best 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 next best move by using MiniMax search (Tuesday, Nov 24th lecture). 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:

```
 *Main> crusher ["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
```

Final Notes:
1) A static board evaluation function is exactly that -- static. It doesn't search ahead. Ever.
2) 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.
3) 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.
4) Before writing any code, play the game a few times.
5) If you are in a one or two person group, the rest of this document does not apply to you. Do not try to implement the interactive part -- there is no bonus in doing extra work. You can do so regardless, but if you're not a 3-person group only your crusher function will be graded.