CSE341 Spring '06 Due Monday, April 10
Assignment 1

In this assignment, you will implement, in ML, a large part of the logic for a basic game of Checkers. If you don't know checkers, you can find full rules on the web,[1] but the basics are: two players called -1 and 1, an 8 x 8 checkerboard with alternate light/dark squares, starting with a dark square in the lower left corner. Label the squares with x, y coordinates in $\{0, \dots, 7\}$, increasing from $(0, 0)$ in the lower left. Each player has pieces of 2 kinds—"men" and "kings". Player 1's men move only towards increasing y; player -1's towards decreasing y. Kings can move in either direction. Moves can be "slides"—to a diagonally adjacent empty square, or "jumps"—over a diagonally adjacent square occupied by an opponent's piece (which is thereby "captured" and removed from the board) to the empty square just past it. You will not have to handle promotion to kings, multiple jumps in one move, nor forced jumps (but see extra credit).

You should represent a piece by a 4-tuple indicating the x, y cordinates of the piece on the board, a ± 1 indicating which player owns the piece, and a bool indicating whether it is a king. Give a type declaration **type piece =** ... for this (Sect 6.1.2, pg 194); you'll be using it repeatedly. A configuration of the board is simply the list of pieces on it; again, use **type config =** ....

Write the following functions. Additional "helper" functions are allowed, if you find them convenient, but should be locally `let`-bound unless you have good reason to make them global.

1. `add_piece` takes a config and a piece and returns a new config containing that piece in addition to the old ones.
2. `on_board` takes an x, y pair and returns `true` if x, y denotes a dark square on the board.
3. `legal_piece` takes a piece and returns `true` if it is legal (on a dark square of the 8x8 board, is owned by one of the players). Use `on_board`.
4. `piece_at` takes a config and an x, y coordinate pair and returns the piece located at that square, if any. Use an *option* (see below and pp. 111-113).
5. `legal_config` tells whether a config is legal: all pieces in it are `on_board`, not duplicated.
6. `legal_move` is the most complex. Given a config c, a piece p, and a direction $dx, dy \in \pm 1$, is it legal for p to move in that direction, and if so, what happens? If it's illegal (e.g., is the wrong direction for a non-king, would move off-board, land on another piece, etc.), then return an empty list. If it is a legal move, then return a one-element list containing the pair (p', q) where p' is the same as p, but with its x, y coordinates updated to reflect the move, and q is a possibly empty list of pieces jumped during the move. Remember that you do not need to handle multiple jumps, so `length(q)<=1`. I've made it a list instead of an option since list extends naturally in to the multi-jump case. Likewise, having `legal_move` return an option is a reasonable alternative, but returning a list simplifies the next function.
7. `all_single_moves` takes a config c and a player $r \in \pm 1$ and returns a list of pairs $(p_i, [m_{i1}, m_{i2}, \dots, m\_ik])$ representing all legal moves for player r starting in config c. Specifically, the p_i will be all of player r's pieces having legal moves, and the corresponding list of m_{ij} will be the different legal moves for p_i, in the format returned by `legal_move`. ($k \leq 4$, since we don't consider multiple jumps. You should *not* implement the forced-jump rule.)

[1]E.g., http://www.chesslab.com/rules/checkersbasics.html

Options: It is generally bad ML style to use a list (which can hold any number of elements) when you will always want zero or one elements. Instead, ML artists prefer "options":

- If `t` is a type, then `t option` is a type (just like `t list` is a type).
- To make a value of type `t option`, write `NONE` (for "zero" elements), or `SOME e` (for the "one" element `e`), where `e` has type `t`.
- The function `isSome` evaluates to `true` if and only if its argument has the form `SOME e`. The function `valOf(x : t option)` evaluates to `e` if `x` is `SOME e`, raising an exception if `x` is `NONE`.

Type Summary: A correct assignment will generate the following bindings.

```
val add_piece = fn : config * piece -> config
val piece_at = fn : config * (int * int) -> piece option
val on_board = fn : int * int -> bool
val legal_piece = fn : piece -> bool
val legal_config = fn : config -> bool
val legal_move = fn : config * piece * int * int -> (piece * piece list) list
val all_single_moves = fn : config * int -> (piece * (piece * piece list) list) list
```

(Your bindings may differ superficially, e.g. showing `config` vs `piece list` or other synonymous types.) Of course, generating these bindings does not guarantee that your solution is correct. Test your functions! My solution is about 100 lines.

Assessment: Your solutions should be

- correct,
- in good style, including indentation and line breaks, and
- written using the features presented in class. In particular, you should not use material outside of chapters 1-3 except as indicated above (especially not references or arrays), and use the =, > and < operators only to compare `int` or `string` expressions. *Do* include types in parameter lists of the functions you define. We will drop this in later assignments, but I think it's useful to type them explicitly in the first assignment.

Syntax Errors, etc.: Small typos can lead to strange error messages. Keep a close watch on your parentheses, and note the following examples.

- `int * int list` means `int * (int list)` and not `(int*int) list`.
- `fun f x : t` means the result type of `f` is `t`, whereas `fun f (x : t)` means the argument type of `f` is `t`. There is no need to write in result types (and in later homeworks no need to write in argument types).
- `fun f (x t)`, `fun f (t x)`, and `fun f (t : x)` are all wrong, but the error messages suggest you are trying to do something more advanced than what you are (which is trying to write `fun f (x : t)`).
- If you start seeing # chars in REPL, take a look at `../help/sml-repl.html`.

Extra Credit: Some or all of: multiple jumps, forced jump rule, coronation, updating a config to reflect a move, a strategy for selecting moves so as to actually play a full game (expert play not required!). Remember, EC is more work than points; get your basic solution running first.