University of Washington
Computer Science & Engineering

CSE 341, Sp '07: Homework #1, Due Saturday, 4/14, 5:00PM

You'll be implementing a portion of the logic for the board game "Othello." To start, you should skim over the rules.

Othello is played by adding pieces to an 8 x 8 board. Pieces don't move after being placed, but they do switch colors (black <--> white).

You will represent board positions as pairs of integers in the range 1..8, with position 1,1 being the lower left corner. (Note that this differs from the indexing shown in the examples on the rules page.) Each square on the board will have a color: ~1 for a black piece, +1 for a white piece (and in some contexts, 0 for unoccupied squares).

We probably will NOT do this in future ML assignments, but for this assignment, I think it is important to be very explicit about types, including explicitly specifying the types of most function arguments (but not function return types). For this purpose, I suggest you make fairly liberal use of ML `type` definitions, including those shown under "Type summary" below.

Perhaps the key definition is the one for *state*: the state of a game tells you whose turn it is and gives a *list* of all pieces (where/what color is each). Yes, I agree that using a 2-d array might seem more natural, but then you wouldn't get much practice with lists, would you? Use a list.

I have various specific functions I want you to write, detailed below. You will probably need additional auxiliary functions to accomplish these tasks. They are certainly allowed, but follow good style in not making them global without good reason; most will be better defined using local `let` bindings within the functions listed. The later functions below can/should make use of earlier ones.

1. `legalpos`: given a position, is it legal, i.e. on the board?
2. `color`: given a position and a state, return the color of the piece in that position; 0 if unoccupied, or off board.
3. `emptysquare`: given a position and state, is that board position empty?
4. `flanks`: given an (empty) position, state, and direction, *if* the next move placed a piece on the given position, a run of zero or more of the opponents pieces will be *outflanked* in the given direction. (See rules.) This function tells how many.
5. `move`: given an (empty) position and a state, return the state that would result from placing a piece on the indicated position. This state should be identical to the given state, except that one more piece is on the board, all runs of opposing pieces that were flanked by this move should have had their colors flipped, and it is now the other player's turn.
6. `show`: print out a state. E.g., given the initial game state

```
val init = (~1,[((4,4),~1),((4,5),1),((5,4),1),((5,5),~1)]):state;
```

the call `show(init);` will print something like configuration shown on the left below:

```
* to move; board:
. . . . . . . .
. . . . . . . .
. . . . . . . .
. . . O * . . .
. . . * O . . .
. . . . . . . .
. . . . . . . .
. . . . . . . .
```

```
O to move; board:
. . . . . . . .
. . . . . . . .
. . O . . . . .
. . * O * . . .
. . . * * . . .
. . . . * . . .
. . . . . . . .
. . . . . . . .
```

```
        val it = () : unit                |
```

Test cases: at the end of your code file, include the `init` binding shown above, followed by the three moves shown below.

```
val next1 = move((3,5),init);   show(next1);
val next2 = move((3,6),next1);  show(next2);
val next3 = move((5,3),next2);  show(next3);
```

This should produce the configuration shown on the right above. A few additional examples are available here. (You should of course test your code more extensively, as we will.)

Type Summary: Your functions should have the following types (approximately; e.g. if "color" shows up as "int" or other type synonym somewhere, it's not a problem).

```
type color = int;                (* color of piece/player: -1 = black, 1  = white; 0 = none *)
type position = int * int;       (* x,y; 1..8 by 1..8, origin in lower left *)
type piece = position * color;
type pieces = piece list;
type state = color * pieces;     (* game state: player moving next, plus all pieces on board *)
type direction = int * int;      (* -1,0,+1 by -1,0,+1 , excluding 0,0 *)

val legalpos = fn : position -> bool
val color = fn : position * state -> color
val emptysquare = fn : position * state -> bool
val flanks = fn : position * state * direction -> int
val move = fn : position * state -> state
val show = fn : state -> unit
```

Grading criteria: I am *not* worried about run-time efficiency. I *am* interested in correctness, clarity, simplicity and ML style -- use let bindings, pattern matching, list operations, etc. as we've been doing in lecture, and use indentation, line breaks, comments, etc. to help make your code intelligible. Use pattern matching wherever possible in preference to if's. "Deep patterns" help. Do not use arrays, refs, while loops or other advanced topics not covered in the reading. My solution runs about 120 lines, with `move` being about 25 (fairly repetitive) lines and the other functions being much shorter. You don't need to duplicate that, but if you find your solution being much longer, think about alternative strategies and/or come talk to us.

Extra Credit: Extend it so that it plays the game, say, as the White player. Read Black moves (or pass) from input, make sure they're legal, then choose among the legal White responses, etc. Maybe detect end of game. You might like to read about "minimax search" or "alpha-beta pruning." This is deliberately open-ended; do as much as you choose. Recall that extra credit benefit, gradewise, is small in proportion to the effort involved. Complete the basic assignment first, and clearly separate, label and *document* the extra credit portion in your solution.

Turnin Instructions: Give us a single .sml file, including at least the test cases above. If you provide additional test cases, clearly separate them from the required cases and document them. Include YOUR NAME in a comment near THE RIGHT MARGIN at the front of your file. Turn in your file electronically via the Catalyst link on the course home page.

Computer Science & Engineering
University of Washington
Box 352350
Seattle, WA 98195-2350
(206) 543-1695 voice, (206) 543-2969 FAX