CSE 341, Fall 2004, Assignment 1
Due: Friday 8 October, 9:00AM

Last updated: September 13

You will write 8 SML functions having to do with Tetris pieces. The sample solution is roughly 75 lines. *See page 2 for additional instructions.*

All necessary Tetris knowledge is in this paragraph; ask if something is unclear. For this homework, a *piece* is a list of *squares*. A *square* is a 1 x 1 area on the two-dimensional plane. We represent a square by its lower-left corner. As examples, $(0,0)$ represents the square with corners $(0,0)$, $(0,1)$, $(1,1)$, and $(1,0)$ and $(0,\tilde{}1)$ represents the square just below the previous example. From now on, we say "a square (x,y)" when we mean "a square represented by (x,y)". In classic Tetris, a piece is four contiguous squares with no repeats, but for this homework, a piece can have any number of squares, they are not necessarily contiguous, and there may be repeats.

1. Write a function `quadrant` that takes a square (x,y) and evaluates to the quadrant the square is in. (Recall that starting on the positive x-axis and proceeding counterclockwise, the quadrants are 1, 2, 3, and 4.)

2. Write a function `on_horizontal_line` that takes a piece p and an integer i and evaluates to true if and only if every square in p has a height (y-coordinate) of i. Hint: A piece with no squares is on every horizontal line. A larger piece is on the line if its first square has height i and the rest of the piece is on the line.

3. Write a function `make_horizontal_line` that takes 3 `int` arguments (a `height`, a `left`, and a `right`) and evaluates to a piece that is a horizontal line of height `height` with exactly the squares between `(left,height)` and `(right,height)` inclusive. Hint: If `left` is greater than `right` there are no such squares, so the empty list is the correct answer.

4. Write a function `has_square` that takes a piece and a square and evaluates to true if and only if the square is in the piece. Hint: No square is in the piece with no squares. Two squares are equal if their x-coordinates are equal and their y-coordinates are equal.

5. Write a function `more_or_same_squares` that takes two pieces and evaluates to true if and only if every square in the second piece is in the first piece. Hint: The function `has_square` will prove useful.

6. Write a function `rightmost_column` that takes a piece p and evaluates to an `int option`. It evaluates to `NONE` if the p has no squares and `SOME i` if p has squares and i is the greatest x-coordinate of any piece in the list. Hint: If a piece is `(x,y)::lst`, then the rightmost-column is `SOME x` if the rightmost-column of `lst` is `NONE` or `SOME i` where `i<x`.

7. Write a function `leftmost_column` that takes a piece p and evaluates to an `int option`. It evaluates to `NONE` if the p has no squares and `SOME i` if p has squares and i is the least x-coordinate of any piece in the list.

8. Write a function `is_horizontal_line` that takes a piece p and evaluates to true if the piece is a contiguous (no gaps), horizontal segment of squares. The piece with no squares is *not* such a piece. Hint: With all your hard work on earlier problems, `is_horizontal_line` should not itself be recursive; p is a horizontal line if the following are true:
   - All p's squares are on the same horizontal line as p's first square.
   - If q is the horizontal line from p's leftmost-column to p's right-most column, then every square in q is in p and every square in p is in q.

9. (**Extra Credit**) Write a function `is_own_mirror_image` that takes a piece *p* and evaluates to true if it would look the same in a mirror. That is, rotating *p* 180 degrees along the vertical line halfway between its leftmost and rightmost square would produce the same piece. Hint: To create a piece *q* that is the mirror image of *p*, each square in *p* moves to its mirror image position in *q*. A square that was *i* columns to the right of the leftmost column ends up *i* columns to the left of the rightmost column. The sample solution uses two helper functions.

   Note: Remember the course policy on extra credit.

Type Summary: Evaluating a correct homework solution should generate these bindings:

```
val quadrant = fn : int * int -> int
val on_horizontal_line = fn : (int * int) list * int -> bool
val rightmost_column = fn : (int * int) list -> int option
val leftmost_column = fn : (int * int) list -> int option
val make_horizontal_line = fn : int * int * int -> (int * int) list
val has_square = fn : (int * int) list * (int * int) -> bool
val more_or_same_squares = fn : (int * int) list * (int * int) list -> bool
val is_horizontal_line = fn : (int * int) list -> bool
```

Of course, generating these bindings does not guarantee that your solutions are correct: *Test your functions.*

Assessment: Your solutions should be:

- Correct
- In good style, including indentation and line breaks
- Written using the features we have used in class. In particular, you must not use references or arrays. (Why would you?) Also, use the `=` operator only to compare to `int` expressions.

Turn-in Instructions

- Put all your solutions in one file, `lastname_hw1.sml`, where `lastname` is replaced with your last name.
- The first line of your `.sml` file should be an ML comment with your name and the phrase `homework 1`.
- Email your solution to `daverich@cs.washington.edu`.
- The subject of your email should be *exactly* `[cse341-hw1]`.
- Your `.sml` file should be an *attachment*.

Syntax Hints

Small syntax errors can lead to strange error messages. Here are 3 examples for function definitions:

1. `int * int list` means `int * (int list)`, not `(int * int) list`. You want `(int * int) list` for a piece.
2. `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 result types (and in later homeworks, no need to write argument types).
3. `fun (x t)`, `fun (t x)`, or `fun (t : x)` are all wrong, but the error message suggests you are trying to do something much more advanced than you actually are (which is trying to write `fun (x : t)`.