The overall objective of this assignment is for you to continue to gain experience with OCaml, adding in some complexity in terms of nested functions (closures) and tail recursion. All the problems require relatively little code ranging from 2 to 10 lines. If any function requires more than that, you can be sure that you need to rethink your solution. Download the handout from autolab and add your function definitions and comments to the misc2.ml file (it contains skeleton OCaml functions) Your task is to replace the text in those files with the appropriate OCaml code for each of those expressions. Do NOT change any of the function headers and be sure to write each function to the exact specs of the lab, including name. Your code will be auto tested, and any deviations will cause your code to fail testing.

To load your code, while in the utop or ocaml interpreter (run from the same directory as misc.ml), type:

```
#use "misc2.ml";
```

**Note:** All the solutions must be done using the purely functional fragment of OCaml, using constructs covered in class, and most require the use of recursion. Solutions using imperative features such as references, arrays, or while loops will receive no credit. Do not use any additional libraries, just use standard OCaml as installed in the Linux and Windows labs.

**Grading Note:** Correctness testing is being done by autolab (that is, does your function have the right types and does it yield the correct answers). However, the specs for this lab are more specific, in terms of when to use pattern matching, when to use if/then/else, and the types of recursion. These things are not being checked in autolab at the moment, so, I will be checking the specs before recording grades. So the autolab grade is not the “final answer” here — if you don’t follow the specs, expect to lose points even if the function runs correctly.

Each problem should bind only one symbol at the top level. Do not litter the global namespace with helper functions. Nest them.

1. (6 pts) Write a simply-recursive (not tail recursive) function to compute factorials (i.e. factorial 10 = 10 * 9 * 8 * ... * 2 * 1). Use an if/then/else statement in the function. Call this function `factorial1`. Have the factorial function handle all numbers from 0 onwards (you don't have to check for less than zero).

   ```ocaml
   # factorial1 4;;
   - : int = 24
   ```

2. (7 pts) Rewrite the previous function using pattern matching instead of if/then/else (this should still be simply recursive not tail recursive). Call this function `factorial2`.

3. (7 pts) Write a new factorial function using pattern matching which is tail-recursive (i.e. it doesn’t cause the stack to grow and there are no pending operations). You will need an internal helper function. Call this `factorial3`.

4. (15 pts) Write a function to compute fibonacci numbers (in the sequence 0, 1, 1, 2, 3, 5, ... where each number is the sum of the previous two numbers on the list). Use pattern matching
(no if/then/else statements). Use a tail recursive solution to make sure the function doesn't take exponential time. Call your function fibonacci. A call to fibonacci should return the nth number in the sequence where n is 0-based.

```
# fibonacci 3;;
- : int = 2
```

Test your function by computing the 44th fibonacci number. What happens when you try to compute larger fibonacci numbers? Why? Indicate the answer in a comment in your code.

5. (15 pts) Write a function sqrt that takes a tolerance tol and a floating-point number x and computes the square root of that number to within the tolerance specified. The function should look like this:

```ocaml
let sqrt tol x = (* your code here *)
```

and should have the type signature:

```
val sqrt : float -> float -> float = <fun>
```

Don't use a library function to compute square roots. Instead, use this algorithm: for an input x and a candidate square root y, compute a better approximation by:

```
next(y) = average(y, x/y)
```

This is not OCaml notation, of course. Repeat this process until the approximation squared is close enough to x, with "close enough" specified by the first argument (tol). Note that the function to compute absolute values of floating-point values is called abs_float.

6. (5 pts) Write a specialized version of the previous function called sqrt2 which will compute square roots to a tolerance of 0.00001. Use the previous function to define this function. Don't refer to the input variable x in your function definition. This is a one-liner.

7. (10 pts) Write a simply-recursive function (not tail recursive) called map that takes two arguments (a function and a list) and returns a new list consisting of the function applied to each member of the original list. There is a built-in map function. You may not use it.

```
# map sqrt [4.0; 16.0; 25.0];;
- : float list = [2.; 4.; 5.]
```

8. (10 pts) Write another mapping function called map2 which is identical to map except that it is tail-recursive. You may find your rev function from the first lab to be useful (make sure your rev function is also tail recursive).

9. (5 pts) Write a function called range which takes two integers a and b as arguments and returns a list of all the integers in the range [a, b] (including a and b). If b is less than a, an empty list should be returned.

```
# range 2 5;;
- : int list = [2; 3; 4; 5]
```
10. (5 pts) Use your `map`, `sqrt2` and `range` functions to compute a value called `roots` which is a list of the square roots of the numbers 1 through 20 to a tolerance of 0.00001 (expressed as floats). You will need the built-in function `float_of_int` which converts integers to floats. You can also do this in one line. Note that `roots` is not a function (it has no parameters).

Submission: Login to autolab and select the Lab 2 assignment. Upload and submit your revised version of misc2.ml.

Attribution: This assignment was adapted from CalTech’s CSE 11 course.