Homework 2

Due Wednesday, April 13, at 11:30pm

Turn in your homework via the course web page as an updated version of the `hw2.ml` text file that I have provided.

Make sure the file can be successfully loaded into the OCaml interpreter via the `#use` directive; if not you get an automatic 0 for the homework!

Recall the CS131 Academic Honesty Policy! You must say whom you discussed the assignment with at the top of your assignment, and also what other resources you used.

For this assignment, you will get practice using higher-order functions in ML. You should always use a higher-order function like `List.map` or `List.fold_right` whenever possible, instead of manually implementing the recursion. In addition, you should obey our usual style rules:

- *Never* use imperative features like assignment and loops. If you're using a construct not discussed in class or in the book, you're probably doing something bad!
- Use pattern matching instead of conditionals wherever it is possible to do so.
- Use local variables to avoid recomputing an expression that is needed multiple times in a function.
- Similarly, avoid code duplication. If the same code is needed in multiple places, possibly with slight variations, make a helper function so that the code only has to be written once.

A few other tips:

- Remember the key to solving problems with recursion: Assume you have the result of the recursive call, and then figure out how to turn that result into the overall answer you desire.
- Create any number of helper functions as needed. It may be advantageous to make these functions local to the main function being defined, so they can refer to names bound in the enclosing function. Try to find opportunities to make use of this feature.
- Write comments where useful to tell the reader what's going on. Comments in ML are enclosed in `(*` and `*)`. The grader should be able to easily understand what your code is doing. One useful comment is to provide the type of any helper function that you define.
- Test your functions on several inputs, including corner cases -- we will be doing the same.

Now on to the assignment! I've provided a file `hw2.ml` that declares all the functions you must implement along with their expected types. Right now each function simply throws an exception (using ML's `raise` expression) that I've defined called `ImplementMe` whenever the function is invoked. Your job is to modify each function's body to do the right thing. **If you modify the function's name or type, you are doing something wrong!**

1. Re-implement the function `allPos` of type `int list -> int list` from Homework 1, but now using a higher-order function from the `List` module rather than using explicit recursion.

2. Re-implement the function `tails` of type `'a list -> 'a list list` from Homework 1, but now using a higher-order function from the `List` module rather than using explicit recursion.

3. In class we saw the `filter` function, of type `('a -> bool) -> 'a list -> 'a list`. Implement `filter` using `List.fold_right`.

4. A useful variant of `map` is the `map2` function, of type `('a * 'b -> 'c) -> 'a list -> 'b list -> 'c list`, which is like `map` but works on two lists instead of one. For example, `map2 (fn (x,y) => x*y) [1,2,3] [4,5,6]` is equal to `[1*4,2*5,3*6]`, which is `[4,10,18]`. Define the `map2` function using explicit recursion. You may assume that the two argument lists have the same length.

5. Now define the `zip` function that we saw in class, of type `'a list * 'b list -> ('a * 'b) list`, using your implementation of `map2`.

6. We've seen two ways to define a two-argument function in ML: the arguments can either be supplied as a tuple, or they can be supplied separately through currying. For example, a function having two integer inputs could be written to have either the type `int * int -> int` or the type `int -> int -> int`. In different circumstances, one or the other form of function may be more convenient. It turns out that a function defined in either form can be converted to the other.

   Define a function `curry` of type `('a * 'b -> 'c) -> ('a -> 'b -> 'c)` and a function `uncurry` of type `('a -> 'b -> 'c) -> ('a * 'b -> 'c)` that perform the conversions. Note: The `->` operator is right-associative, so the types of `curry` and `uncurry` can be equivalently written as `('a * 'b -> 'c) -> 'a -> 'b -> 'c` and `('a -> 'b -> 'c) -> 'a * 'b -> 'c`, respectively.

   For example, suppose `f` has type `int * int -> int`, and let `g` be `curry f`. Then `g e1 e2` should yield the same answer as `f(e1,e2)`, for all arguments `e1` and `e2`. The `uncurry` function performs the reverse transformation.

7. Implement a function `primesUpTo` of type `int -> int list` that uses the ancient Sieve of Eratosthenes method to compute a list of the prime numbers from least to greatest between 2 and the given number (which you may assume to be greater than or equal to 2).

   Given an integer `n`, first produce a list of the integers between 2 and `n`, inclusive. I've provided a helper function `intsBetween` that you should implement for this purpose. We call the resulting list the *candidate list*. Now implement the helper function `sieve` of type `int list -> int list`, which should take the candidate list and return the list of primes. The `sieve` function works as follows (and check out the graphical illustration on Wikipedia). The first element of the candidate list is prime. Call this element `p1`. Remove from the candidate list all multiples of `p1` (including `p1` itself), to get the new candidate list. Again, the first element of this new candidate list, call it `p2`, is prime. Again, remove all multiples of `p2` from the candidate list. Continue in this way until the candidate list is empty. While I've described this algorithm as imperatively modifying the candidate list, of course in ML you should implement this in the natural functional way and using higher-order functions where possible.

8. Implement a function `ccipher` of type `int -> ((string -> string) * (string -> string))` that implements a Caesar cipher, which is a simple form of cryptography used by Julius Caesar. The idea is simply to shift each character up by a specified number of characters. For example, encrypting the string `"lazy"` using the shift value 3 yields the encrypted string `"odcb"`. Notice how the shifting wraps back around to the beginning of the alphabet. Then decrypting `"odcb"` with the shift value 3 gives you back the original string.

   Your function `ccipher` should take a shift value, which is assumed to be an integer between 1 and 25 inclusive, and produce a pair of two functions, which are respectively the encryption and decryption functions specialized to the given shift value. For example:

```
let (encr, decr) = ccipher 3 in encr "lazy"
```

yields the value `"odcb"`.

You may assume that strings to be encrypted consist only of the 26 lowercase letters of the English alphabet.

I have provided functions `str2charList` and `charList2str`, which convert between a string and a list of characters and may be useful for you. The library function `Char.code`, which converts a character to its ASCII integral value, and its inverse `Char.chr`, may also be of use.