# Homework 2

## Due Monday, October 20, 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 list 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 with higher-order functions in ML. In a few places below, you are required to implement your functions in a particular way, so *pay attention to those directives or you will get no credit for the problem*. 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 natural 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:

- Create any number of helper functions as needed, except where explicitly disallowed. 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 OCaml 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. One useful approach is *test-driven development*: write a bunch of test cases first, before you start coding, to act as a "spec" for what you need to do.

Now on to the assignment! I've provided a file `hw2.ml` that has a comment giving the name and type of each function that you must implement. **Make sure each of your functions has exactly the expected name and type; otherwise you will get no credit for it.** I've also declared some datatypes that are useful in certain problems.

### Problem #1: Using higher-order functions

Implement each function below such that the *entire* function body is a single call to one of the following functions from the `List` module: `map`, `filter`, `fold_right`, `fold_left`.

a. A function `doubleAllPos` of type `int list -> int list` that returns the identical list to the original one but with all positive integers doubled in value. Other integers are kept but are unchanged. For example, `doubleAllPos [1;2;-1;4;-3;0]` returns `[2;4;-1;8;-3;0]`.

b. The function `unzip` of type `('a * 'b) list -> ('a list * 'b list)` from class, which turns a list of pairs into a pair of lists. For example, `unzip [(1,2);(3,4)]` returns `([1;3], [2;4])`.

c. The function `encode` from the last homework.

d. The function `intOfDigits` from the last homework.

## Problem #2: Defining higher-order functions

a. 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 (fun 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. *Do not use any functions from the `List` module or other modules.*

b. Now implement the `zip` function of type `('a list * 'b list) -> ('a * 'b) list` as a function whose entire body is a single call to `map2`. For example, `zip ([1;3], [2;4])` returns `[(1,2);(3,4)]`.

c. A useful variant of `fold_right` and `fold_left` is the `foldn` function, of type `(int -> 'a -> 'a) -> int -> 'a -> 'a`, which folds over an integer (assumed to be nonnegative) rather than a list. In particular, `foldn f n b` should return the value of `f n (f (n-1) (f (n-2) ... (f 1 b)))`. For example, `foldn (fun x y -> x*y) n 1` computes the factorial of `n`. Implement `foldn` using explicit recursion.

d. Now implement the `clone` function from the last homework as a function whose entire body is a single call to `foldn`.

## Problem #3: Dictionaries

A *dictionary* (sometimes also called a *map*) is a mapping from keys to values, supporting three main operations: `empty`, which returns an empty dictionary; `put`, which adds a new key-value pair to a given dictionary; and `get`, which looks up the value associated with a given key in a given dictionary. If the given key is already mapped to some value in the dictionary, then `put` should (conceptually) replace the old key-value pair with the new one. To handle the case when the given key is not mapped to some value in the dictionary, `get` will return an `option` (we saw that type in class), i.e. either the value `None` or the value `Some v`, where `v` is the value mapped to the given key in the dictionary.

In this problem we'll explore three different implementations of a dictionary data structure. *It's fine if the types that OCaml infers for some of these functions are more general than the types we specify. Specifically, the inferred types could use a type variable like `'a` in place of a more specific type.*

a. Our first implementation of a dictionary is as an *association list*, i.e. a list of pairs. Implement `empty1`, `put1`, and `get1` for association lists (we use the suffix `1` to distinguish from other implementations below). As an example of how this representation of dictionaries works, the dictionary that maps `"hello"` to `5` and has no other entries is represented as `[("hello",5)]`. To get the effect of replacing old entries for a key, `put1` should simply add new entries to the front of the

list, and accordingly `get1` should return the leftmost value whose associated key matches the given key.

b. A different way to implement a dictionary is by declaring a new datatype:

```
type ('a,'b) dict2 = Empty | Entry of 'a * 'b * ('a,'b) dict2
```

Here `dict2` is *polymorphic* over the key and value types, which respectively are represented by the *type variables* `'a` and `'b`. For example, the dictionary that maps `"hello"` to `5` and has no other entries would be represented as the value `Entry("hello", 5, Empty)` and would have type `(string,int) dict2`.

Implement `empty2`, `put2`, and `get2` for `dict2`. As above, new entries should be added to the front of the dictionary, and `get2` should return the leftmost match.

c. Conceptually a dictionary is just a function from keys to values. Since OCaml has first-class functions, we can choose to represent dictionaries as actual functions. We define the following type:

```
type ('a,'b) dict3 = ('a -> 'b option)
```

We haven't seen the above syntax before (note that the right-hand side just says `('a -> 'b option)` rather than something like `Foo of ('a -> 'b option)`). Here `dict3` is a *type synonym*: it is just a shorthand for the given function type rather than a new type. As an example of how this representation of dictionaries works, the following dictionary maps `"hello"` to `5` and has no other entries:

```
(function s ->
  match s with
    "hello" -> Some 5
  | _ -> None)
```

One advantage of this representation over the two dictionary implementations above is that we can represent infinite-size dictionaries. For instance, the following dictionary maps all strings to their length (using the `String.length` function):

```
(function s -> Some(String.length s))
```

Implement `empty3`, `put3`, and `get3` for `dict3`. *It's fine if the types that OCaml infers for these functions use `('a -> 'b option)` in place of `('a,'b) dict3`, since they are synonyms for one another.*