Important notes about grading:

- **Compiler errors:** In cases where programs cannot be successfully compiled, a grade of zero will be assigned. If you encounter difficulties in compiling your assignment, we encourage you to seek assistance by asking questions on Piazza, attending recitation sessions or consulting during office hours.
- You may rename function arguments, but do not modify function names. Doing so will cause you to fail the tests.
- You must not use any mutation operations of OCaml for any of these questions: no arrays, for- or while-loops, references, etc.

Compile and run your code:

After downloading and unzipping the file for assignment1, in your terminal, `cd` into the directory which contains `src`. You should write your code in `src/assignment2.ml`.

Use `ocamlc -o test src/assignment2.ml` to compile the code. Then, use `./test` to excute it.

About library functions:

You can use any library functions from the OCaml List module for this assignment.

Submission:

Please submit `assignment2.ml` to Canvas.

Problem 1

Write a recursive function

```
cond_dup : 'a list -> ('a -> bool) -> 'a list
```

that takes in a list and a preciate and duplicates all elements which satisfy the condition expressed in the predicate.

In [ ]:
```
let rec cond_dup l f =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (cond_dup [3;4;5] (fun x -> x mod 2 = 1) = [3;3;4;5;5]);
assert (cond_dup [] (fun x -> x mod 2 = 1) = []);
assert (cond_dup [1;2;3;4;5] (fun x -> x mod 2 = 0) = [1;2;2;3;4;4;5])
```

Problem 2

Write a recursive function

```
n_times : ('a -> 'a) * int * 'a -> 'a
```

such that `n_times (f,n,v)` applies `f` to `v` `n` times. If `n<=0` return `v`.

In [ ]:
```
let rec n_times (f, n, v) =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (n_times((fun x-> x+1), 50, 0) = 50);
assert (n_times ((fun x->x+1), 0, 1) = 1);
assert (n_times((fun x-> x+2), 50, 0) = 100)
```

Problem 3

Write a recursive function

```
zipwith : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

such that `zipwith f l1 l2` generates a list whose `ith` element is obtained by applying `f` to the `ith` element of `l1` and the `ith` element of `l2`. If the lists have different lengths, the extra elements in the longer list should be ignored.

In [ ]:
```
let rec zipwith f l1 l2 =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (zipwith (+) [1;2;3] [4;5] = [5;7]);
assert (zipwith (fun x y -> (x,y)) [1;2;3;4] [5;6;7] = [(1,5); (2,6); (3,7)])
```

Problem 4 (Hard)

Write a function

```
buckets : ('a -> 'a -> bool) -> 'a list -> 'a list list
```

that partitions a list into equivalence classes. That is, `buckets equiv lst` should return a list of lists where each sublist in the result contains equivalent elements, where two elements are considered equivalent if `equiv` returns `true`. For example:

```
buckets (=) [1;2;3;4] = [[1];[2];[3];[4]]
buckets (=) [1;2;3;4;2;3;4;3;4] = [[1];[2;2];[3;3;3];[4;4;4]]
buckets (fun x y -> (=) (x mod 3) (y mod 3)) [1;2;3;4;5;6] = [[1;4];[2;5];[3;6]]
```

The order of the buckets must reflect the order in which the elements appear in the original list. For example, the output of `buckets (=) [1;2;3;4]` should be `[[1];[2];[3];[4]]` and not `[[2];[1];[3];[4]]` or any other permutation.

The order of the elements in each bucket must reflect the order in which the elements appear in the original list. For example, the output of `buckets (fun x y -> (=) (x mod 3) (y mod 3)) [1;2;3;4;5;6]` should be `[[1;4];[2;5];[3;6]]` and not `[[4;1];[5;2];[3;6]]` or any other permutations.

Assume that the comparison function `('a -> 'a -> bool)` is commutative, associative and idempotent.

You are not allowed to use sets or hash tables in your solution.

The list concatenation operator `@` may come in handy. Feel free to use helper functions.

In [ ]:
```
let buckets p l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (buckets (=) [1;2;3;4] = [[1];[2];[3];[4]]);
assert (buckets (=) [1;2;3;4;2;3;4;3;4] = [[1];[2;2];[3;3;3];[4;4;4]]);
assert (buckets (fun x y -> (=) (x mod 3) (y mod 3)) [1;2;3;4;5;6] = [[1;4];[2;5];[3;6]]);
```

Tail Recursion

Problem 5

The usual recursive formulation of fibonacci function

```
let rec fib n =
  if n = 0 then 0
  else if n = 1 then 1
  else fib (n-1) + fib (n-2)
```

has exponential running time. It will take a long time to compute `fib 50`.

But we know that fibonacci number can be computed in linear time by remembering just the current `cur` and the previous `prev` fibonacci number. In this case, the next fibonacci number is computed as the sum of the current and the previous numbers. Implement a tail recursive function `fib_tailrec` that uses this idea and computes the `nth` fibonacci number in linear time.

```
fib_tailrec : int -> int
```

In [ ]:
```
let fib_tailrec n =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (fib_tailrec 50 = 12586269025);
assert (fib_tailrec 90 = 2880067194370816120)
```

Map and Fold

For the following problems, you must use `List.map`, `List.fold_left`, or `List.fold_right` to complete the following functions. You are not allowed to use the "rec" keyword in your solution. A solution, even a working one, that uses explicit recursion via "rec" will receive no credit. You may write useful auxiliary functions; they just may not be recursive. The goal of this part is to get you think about writing programs in a different style from what you are used to. We are trying to enlarge your programming toolkit.

Some of these functions will require just `map` or `fold`, but some will require a combination of the two. The `map/reduce` design pattern may come in handy.

Some of these functions might be simpler to solve using `fold_right` instead of `fold_left`.

Problem 6

Write a function

```
sum_rows: int list list -> int list
```

that takes a list of int lists (call an internal list a "row") and returns a one-dimensional list of ints, each int equal to the sum of the corresponding row in the input.

In [ ]:
```
let sum_rows (rows:int list list) : int list =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (sum_rows [[1;2]; [3;4]] = [3; 7]);
assert (sum_rows [[5;6;7;8;9]; [10]] = [35; 10])
```

Problem 7

Write a function

```
ap: ('a -> 'b) list -> 'a list -> 'b list
```

`ap fs args` applies each function in `fs` to each argument in `args` in order. For example, `ap [(fun x -> x^"?"); (fun x -> x^"!")] ["foo";"bar"] = ["foo?";"bar?";"foo!";"bar!"]` where `^` is an OCaml operator for string concatenation.

In [ ]:
```
let ap fs args =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (ap [(fun x -> x^"?"); (fun x -> x^"!")] ["foo";"bar"] = ["foo?";"bar?";"foo!";"bar!"]);
```

Problem 8

Write a function

```
prefixes: 'a list -> 'a list list
```

`prefixes l` returns a list of all non-empty prefixes of an input list `l`, ordered from shortest to longest. There are no non-empty prefixes of an empty list.

In [ ]:
```
let prefixes l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (prefixes [1;2;3;4] = [[1]; [1;2]; [1;2;3]; [1;2;3;4]]);
assert (prefixes [] = []);
```

Problem 9

Write a function

```
powerset: 'a list -> 'a list list
```

`powerset l` returns the powerset of the set of values in an input list `l` (the power set of a Set `A` is defined as the set of all subsets of the Set `A` including the Set itself). The order in the returned nested list (and each list element within) does not matter.

In [ ]:
```
let powerset l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
# powerset [1;2;3];;
- : int list list = [[1]; [1; 2]; [1; 2; 3]; [1; 3]; [2]; [2; 3]; [3]]

# powerset [];;
- : 'a list list = []
```

Problem 10

Write a function

```
assoc_list: 'a list -> ('a * int) list
```

that that takes a list as input and returns a list of pairs where the first value of each pair is an element of the input list and the second integer of the pair is the number of occurrences of that element in the input list. This associative list should not contain duplicates. The order in the returned list does not matter.

In [ ]:
```
let assoc_list l =
  (* YOUR CODE HERE *)
```

In [ ]:
```
# assoc_list [1; 2; 2; 1; 3];;
- : (int * int) list = [(2,2); (1, 2); (3, 1)]

# assoc_list [true;false;false;true;false;false;false];;
- : (bool * int) list = [(false,5);(true,2)]
```