Important notes about grading:

1. **Compiler errors:** All code you submit must compile. Programs that do not compile will probably receive an automatic zero. If you are having trouble getting your assignment to compile, please visit recitation or office hours. If you run out of time, it is better to comment out the parts that do not compile, than hand in a more complete file that does not compile.
2. **Late assignments:** Please carefully review the course website's policy on late assignments, as all assignments handed in after the deadline will be considered late. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

How to compile and test your solution:

1. **Download assignment1.zip** Your code must be implemented in `src/assignment1.ml`. Provided test cases can be found in `test/public.ml`.
2. **Compile your code:** Under `assignment1/`, you can compile the code and run the tests by typing `dune runtest -f` in the command-line window.
3. **Test your code:** We recommend you write student tests in `test/student.ml`.

Problem 1

Write a function

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

list that takes in a list and preciate and duplicates all elements which satisfy the condition expressed in the predicate. For example, `cond_dup [3;4;5] (fun x -> x mod 2 = 1) = [3;3;4;5;5]`.

In [ ]:

```
let rec cond_dup l f =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

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

Problem 2

Write a function

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

such that `n_times (f,n,v)` applies `f` to `v` `n` times. For example, `n_times((fun x-> x+1), 50, 0) = 50`. If `n<=0` return `v`.

In [ ]:

```
let rec n_times (f, n, v) =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (n_times((fun x-> x+1), 50, 0) = 50)
```

Problem 3

Write a function

```
range : int -> int -> int list
```

such that `range num1 num2` returns an ordered list of all integers from `num1` to `num2` inclusive. For example, `range 2 5 = [2;3;4;5]`. Raise the exception `IncorrectRange` if `num2 < num1`.

In [ ]:

```
exception IncorrectRange

let rec range num1 num2 =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (range 2 5 = [2;3;4;5])
```

Problem 4

Write a 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. For example, `zipwith (+) [1;2;3] [4;5] = [5;7]`.

In [ ]:

```
let rec zipwith f l1 l2 =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (zipwith (+) [1;2;3] [4;5] = [5;7])
```

Problem 5

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.

Just use lists. Do not use sets or hash tables.

List append function `@` may come in handy. `[1;2;3] @ [4;5;6] = [1;2;3;4;5;6]`.

In [ ]:

```
let buckets p l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

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]])
```

Problem 6

Write a function

```
remove_stutter : 'a list -> 'a list
```

that removes stuttering from the original list. For example, `remove_stutter [1;2;2;3;1;1;1;4;4;2;2]= [1;2;3;1;4;2]`. Option type may come in handy.

In [ ]:

```
let remove_stutter l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (remove_stutter [1;2;2;3;1;1;1;4;4;2;2] = [1; 2; 3; 1; 4; 2])
```

Problem 7

Write a function

```
flatten : 'a list list -> 'a list
```

that flattens a list. For example, `flatten [[1;2];[3;4]] = [1;2;3;4]`.

In [ ]:

```
let rec flatten l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (flatten ([[1;2];[3;4]]) = [1;2;3;4])
```

Problem 8

Write a function

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree

fold_inorder : ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a
```

That does a inorder fold of the tree. For example,

`fold_inorder (fun acc x -> acc @ [x]) [] (Node (Node (Leaf,1,Leaf), 2, Node (Leaf,3,Leaf))) = [1;2;3]`

In [ ]:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

```
let rec fold_inorder f acc t =
  (* YOUR CODE HERE *)
  raise (Failure "Not implemented")
```

In [ ]:

```
assert (fold_inorder (fun acc x -> acc @ [x]) [] (Node (Node (Leaf,1,Leaf), 2, Node (Leaf,3,Leaf)))
= [1;2;3])
```

Problem 9

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`. You might have to interrupt the kernel if you did try to do `fib 50` in the notebook.

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. Then the program continues by setting `prev` to be `cur` and `cur` to be `cur + prev`.

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 *)
  raise (Failure "Not implemented")
```

In [ ]:

```
assert (fib_tailrec 50 = 12586269025)
```