Acknowledgement

This recitation is adapted from the excellent coursework from Prof. Clarkson (U. Cornell): Link.

CSE216 Lab on Folding and Tail Recursion

- This work will not be graded but it is a part of the curriculum. You are expected to work as hard as possible.
- Group work is also highly encouraged
- Happy learning!

Exercise 1 Folding

Suppose we want to write a function to sum a list of integers. By now you should be able to write the following code:

```
let rec sum (l : int list) : int =
  match l with
    [] -> 0
  | x :: xs -> x + (sum xs)
```

Now suppose we want to concatenate a list of strings. We can write:

```
let rec concat (l : string list) : string =
  match l with
    [] -> ""
  | x :: xs -> x ^ (concat xs)
```

Notice that both functions look almost identical. With the exception of the different types and different operation (`^` vs `+`), both functions are equivalent. In both cases, we walk down a list performing some operation with the data at each step. Since we like to reuse code in this class, is there some easier way to encode this?

It turns out we can abstract all of the details of traversing a list. The idea is simple. As we walk across a list, we store an accumulator, a value that stores some data that is important to us. For example, if we want to walk across a list of integers and sum them, we could store the current sum in the accumulator. We start with the accumulator set to 0. As we come across each new element, we add the element to the accumulator. When we reach the end, we return the value stored in the accumulator.

Let's try to rewrite the sum function to introduce the idea of an accumulator.

```
let rec sum' (acc : int) (l : int list) : int =
  match l with
```

```
      [] -> acc
    | x :: xs -> sum' (acc + x) xs
```

Of course, to get the sum, we must call `sum'` with 0 for the initial acc value. Similarly, we can rewrite `concat` with this concept of the accumulator.

```
let rec concat' (acc : string) (l : string list) : string =
  match l with
    [] -> acc
  | x :: xs -> concat' (acc ^ x) xs
```

To use this function, we pass in the empty string for the initial accumulator. Now we see even more similarity between the two functions. We are in a position to eliminate any differences between the two by passing in the operator that acts on the head of the list and the accumulator. The result is the most powerful list function in the list library: `List.fold_left`.

```
let rec fold_left (f : 'a -> 'b ->'a) (acc : 'a) (l : 'b list): 'a =
  match l with
    [] -> acc
  | x :: xs -> fold_left f (f acc x) xs
```

Now we can rewrite `sum` and `concat` as

```
let sum (l : int list) : int =
  List.fold_left (fun acc x -> acc + x) 0 l

let concat (l : string list) : string =
  List.fold_left (fun acc x -> acc ^ x) "" l
```

Given a function f of type `'a -> 'b -> 'a, the` expression `List.fold_left f b [x1; x2; ...; xn]` evaluates to `f (... (f (f b x1) x2) ...) xn`. The 'left' in List.fold_left comes from the fact that it walks the list from left to right. Thus, we have to be careful sometimes in applying functions in the right order (hence `acc ^ x` and not `x ^ acc` in concat above). There is also a library function List.fold_right which traverses the list from right to left. Note that `List.fold_right` uses a different argument order than `List.fold_left`. The list and the accumulator arguments are switched, and the function takes in the accumulator as its second curried argument, not its first. So, `List.fold_right f [x1; x2; ...; xn] b` evaluates to `f x1 (f x2 (... (f xn b)...))`. The code for `List.fold_right` is

```
let rec fold_right (f : 'a -> 'b -> 'b) (l : 'a list) (acc : 'b) : 'b =
  match l with
    [] -> acc
  | x :: xs -> f x (List.fold_right f xs acc)
```

We can use `List.fold_right` to write concat in the more natural manner.

let `concat (l : string list) : string = List.fold_right (fun x acc -> x ^ acc) l ""` But we can still do better. Remember, both `List.fold_left` and `List.fold_right` are curried, so that we can leave out the list argument and write

```
let sum = List.fold_left (fun a x -> x + a) 0
let concat = List.fold_left (fun a x -> a ^ x) ""
```

We can do even better. OCaml provides a convenient notation to use infix binary operators as curried functions by enclosing them in parentheses. So `(+)` is a curried function that takes two integers and add them, `(^)` is a curried function that takes two strings and concatenates them. We can write sum and concat one last time as

```
let sum = List.fold_left (+) 0
let concat = List.fold_left (^) ""
```

Now you see the true power of functional programming.

1. Find a function in Java 8's Stream API that is similar to `List.fold_left`. Then write in a single line the code that sums a list of `int`s; in other words, fill in the todo below.

```
// Java 8 code
List<Integer> list = Arrays.asList(1, 2, 3, 4, 5);
int sum =  /* todo */
System.out.println(sum); // Output: 15
```

2. Folding is so general, that we can write many list function in terms of `List.fold_left` or `List.fold_right`. Fill in the todo below.

```
let length l = List.fold_left (*todo*)

let rev l = List.fold_left (*todo*)

let map f l = List.fold_right (*todo*)

let filter f l =
  List.fold_right (*todo*)
```

Exercise 2 Tail recursion

A function is tail recursive if it calls itself recursively but does not perform any computation after the recursive call returns, and immediately returns to its caller the value of its recursive call. Observe the following difference between the sum and sum' functions above: In the first sum function, which is not tail recursive, after the recursive call returned its value, we add x to it. In the tail recursive sum', after the recursive call returns, we immediately return the value without further computation. Notice also that List.fold_left is tail recursive, whereas List.fold_right is not.

Why do we care about tail recursion? Performance. Conceptually, there is a call stack, which is a stack (the data structure with push and pop operations) with one element for each function call that has been started but has not yet completed. Each element stores things like the value of local variables and what part of the function has not been evaluated yet. When the evaluation of one function body calls another function, a new element is pushed on the call stack and it is popped off when the called function completes. When a function makes a recursive call to itself and there is nothing more for the caller to do after the callee returns (except return the callee's result), this situation is called a tail call. Functional languages like OCaml (and even imperative languages like C++) typically include an hugely useful optimization: when a call is a tail call, the caller's stack-frame is popped before the call—the callee's stack-frame just replaces the caller's. This makes sense: the caller was just going to return the callee's result anyway. With this optimization, recursion can sometimes be as efficient as a while loop in imperative languages (such loops don't make the call-stack bigger.) The "sometimes" is exactly when calls are tail calls—something both you and the compiler can (often) figure out. With tail-call optimization, the space performance of a recursive algorithm can be reduced from O(n) to O(1), that is, from one stack frame per call to a single stack frame for all calls.

So when you have a choice between using `List.fold_left` and `List.fold_right`, you are likely better off using `List.fold_left` on really long lists. Nonetheless, it is sometimes easier to express a computation using List.fold_right (notice that we wrote map in terms of fold_right in lecture on map and fold), so there is a tradeoff between performance and code clarity. If you want to use List.fold_right on a very lengthy list, you might instead choose to reverse the list first, then use List.fold_left. For example, we could have written map as follows:

```ocaml
let map f l = List.fold_left (fun a x -> (f x) :: a) [] (List.rev l)
```

1. Compare the following two Python implementations of factorial functions, which one is tail-recursive; which one is not?

```python
# Python code
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)

def factorial2(n, acc=1):
    if n == 0:
        return acc
    else:
        return factorial2(n-1, acc*n)
```

Note: That results in a tail recursive hence memory efficient implementation. But that doesn't mean that a tail-recursive implementation is always better. On small to medium sized lists, the overhead of reversing the list (both in time and in allocating memory for the reversed list) can make the tail-recursive version less time efficient.

2. Write the `filter` function with the help of `List.fold_left`. Using `List.fold_left` instead of `List.fold_left` allows your `filter` function to be tail-recursive.

```
let filter f l =
  List.fold_left (*todo*)
```

3. Write a tail-recursive function `sum_list : int list -> int` that takes a list of integers and returns their sum. Your function should use tail recursion and not use any library functions like `List.fold_left`.

[Hint: You may need a helper function `sum_helper` that takes two arguments: the remaining list to be summed, and the accumulator that keeps track of the running sum.]