Acknowledgement

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

CSE216 Week 12 - Ungraded

- This work will not be graded, but you are expected to work as hard as possible.
- Group work is highly encouraged

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. Write an OCaml function using `List.fold_left` that takes a list of integers and returns the sum of the squares of these integers.

```
let sum_of_squares lst =
  List.fold_left (*todo*)
```

3. Create a function that calculates the weighted average of a list of grades, where each grade is paired with its corresponding weight. For example, `calculate_weighted_average [(90.0, 0.3); (80.0, 0.4); (70.0, 0.3)]` should return the weighted average: `(90.0*0.3 + 80.0*0.4 + 70.0 * 0.3)/(0.3 + 0.4 + 0.3)`.

```
type grade = float * float
let calculate_weighted_average (g: grade list) : float = (*todo*)
```

4. 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*)
```