The overall objective of this assignment is for you to continue to gain experience with OCaml, particularly the important functional programming concept of folding. All the problems require relatively little code ranging from 2 to 10 lines. If any function requires more than that, you can be sure that you need to rethink your solution. Download the handout from autolab and add your function definitions and comments to the misc5.ml file (it contains skeleton OCaml functions) Your task is to replace the ‘failwith “to be implemented”’ text in that file with the appropriate OCaml code for each of those expressions. Do NOT change any of the function headers and be sure to write each function to the exact specs of the lab, including name. Your code will be auto tested, and any deviations will cause your code to fail testing.

Remember that each function you write must be documented with type signature, descriptions and input/result examples.

To load your code, while in the utop or ocaml interpreter (run from the same directory as misc.ml), type:

`#use "misc5.ml"; **Be sure to test your file this way before uploading it to autolab. If utop throws errors, so will autolab.**`

**Grading Note:** Autolab may not detect it if you alter the templates. However, the specs for the assignment say that you must use the templates as they appear in the handout. I will be reviewing code and deducting the points for any problems where the template has been altered! The purpose of this assignment is to get you to start thinking in terms of folding, and the templates enforce that way of thinking.

All the solutions must be done using the purely functional fragment of OCaml, using constructs covered in class, and most require the use of recursion. Solutions using imperative features such as references, arrays, or while loops will receive no credit. Do not use any additional libraries, just use standard OCaml as installed in the Linux and Windows labs. You may use the built-in functions that are listed in the comments of misc5.ml (map, fold_left, etc.).

Each problem should bind only one symbol at the top level. Do not litter the global namespace with helper functions. Nest them or define them inline as anonymous functions as appropriate.

1. (10 pts) Fill in the skeleton given for sqsum, which uses List.fold_left to get an OCaml function sqsum: int list -> int that takes a list of integers [x1;...;xn] and returns the integer: x1^2 + ... + xn^2. Your task is to fill in the appropriate values for (1) the folding function f and (2) the base case base. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```ocaml
# sqsum [];;
- : int = 0
# sqsum [1;2;3;4] ;;
- : int = 30
# sqsum [-1;-2;-3;-4] ;;
- : int = 30
```
2. (15 pts) Fill in the skeleton given for `sepConcat`, which uses `List.fold_left` to get an OCaml function `sepConcat : string -> string list -> -> string`. The function `sepConcat` is a curried function which takes as input a string `sep` to be used as a separator, and a list of strings `[s1;...;sn]`. If there are 0 strings in the list, then `sepConcat` should return `''`. If there is 1 string in the list, then `sepConcat` should return the concatenation `s1 sep s2 sep s3 ... sep sn`. You should only modify the parts of the skeleton consisting of `failwith "to be implemented"`. You will need to define the function `f`, and give values for `base` and `l`. Once you have filled in these parts, you should get the following behavior at the OCaml prompt:

```ocaml
# sepConcat "", " ["foo";"bar";"baz"];
- : string = "foo", bar, baz
# sepConcat "---" [];
- : string = ""
# sepConcat "" ["a";"b";"c";"d";"e"];
- : string = "abcde"
# sepConcat "X" ["hello"];;
- : string = "hello"
```

3. (5 pts) Implement the curried OCaml function `stringOfList : ('a -> string) -> 'a list -> string`. The first input is a function `f : 'a -> string` which will be called by `stringOfList` to convert each element of the list to a string. The second input is a list `l : 'a list`, which we will think of as having the elements `l1, l2, ..., ln`. Your `stringOfList` function should return a string representation of the list `l` as a concatenation of the following: 

"[" (f l1) "," (f l2) "," (f l3) "," ... "," (f ln) "]". This function can be implemented on one line without using any recursion by calling `List.map` and `sepConcat` with appropriate inputs. Once you have completed this function, you should get the following behavior at the OCaml prompt:

```ocaml
# stringOfList string_of_int [1;2;3;4;5;6];
- : string = "[1; 2; 3; 4; 5; 6]"
# stringOfList (fun x -> x) ["foo"];
- : string = "[foo]"
# stringOfList (stringOfList string_of_int) [[1;2;3];[4;5];[6];[[]]];
- : string = "[[1; 2; 3]; [4; 5]; [6]; [[]]]"
```

4. pipe and pipec

   a. (10 pts) Fill in the skeleton given for `pipe`, which uses `List.fold_left` to get an OCaml function `pipe : ('a -> 'a) list -> ('a -> 'a)`. The function `pipe` takes a list of functions `[f1;...;fn]` and returns a function `f` such that for any `x`, the application `f x` returns the result `fn(...(f2(f1 x)))`. Again, your task is to fill in the appropriate values for (1) the folding function `f` and (2) the base case `base`. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```ocaml
# pipe [] 3;;
- : int = 3
# pipe [(fun x -> 2*x);(fun x -> x + 3)] 3 ;;
- : int = 9
# pipe [(fun x -> x + 3);(fun x -> 2*x)] 3;;
- : int = 12
```
b. (10 pts) Fill in the skeleton given for `pipec`, which has the same behavior as above BUT uses function currying. In this version of the function, you will not explicitly take in the second parameter. Instead you will return a function that expects this parameter and uses it when it is received. Once you have implemented the function, you should get the following (same) behavior at the OCaml prompt:

```ocaml
# pipec [] 3;;
- : int = 3
# pipec [(fun x -> 2*x);(fun x -> x + 3)] 3 ;;
- : int = 9
# pipec [(fun x -> x + 3);(fun x-> 2*x)] 3;;
- : int = 12
```

5. (10 pts) Fill in the skeleton given for `prodLists`, which uses `List.fold_left` to get an OCaml function `prodLists : int list -> int list -> int list`. The function `prodLists` takes two lists of integers \([x_1;...;x_n]\) and \([y_1;...;y_n]\) (you can assume that they’re the same length) and returns a list of integers with the product of each pair of numbers \([x_1*y_1;...;x_n*y_n]\). Your task is to fill in the appropriate values for (1) the folding function \(f\) (2) the base case \(\text{base}\) and (3) the \(\text{args}\) being passed to \(\text{fold_left}\). Hint: Look at using `List.combine`. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```ocaml
# prodLists [] [];;
- : int list = []
# prodLists [3] [4];;
- : int list = [12]
# prodLists [10; 20; 30] [5; 5; 5];;
- : int list = [50; 100; 150]
```

6. Big Numbers: As you may have noticed, the OCaml type `int` only contains values up to a certain size. For example, entering `99999999999999999999` results in the message “Integer literal exceeds the range of representable integers of type int”. The next questions involve implementing functions to manipulate large numbers represented as lists of integers.

Several helper functions (`clone`, `padZero`, `removeZero`) have been included in the handout file. Use these in your implementations wherever they are helpful.

a. (20 pts) Warning: this one is significantly tougher than the earlier ones! Fill out the implementation for `bigAdd : int list -> int list -> int list`, so that it takes two integer lists, where each integer is in the range \([0..9]\) and returns the list corresponding to the addition of the two big integers. Again, you have to fill in the implementation to supply the appropriate values to \(f\), \(\text{base}\), \(\text{args}\). One of the keys here is to realize that your accumulator can carry more than one piece of data. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```ocaml
# bigAdd [9;9] [1;0;0;2];;
- : int list = [1;1;0;1]
# bigAdd [9;9;9;9] [9;9;9];;
- : int list = [1;0;9;9;8]
```
b. (20 pts EXTRA CREDIT) Next you will write functions to multiply two big integers. First write a function `mulByDigit : int -> int list -> int list` which takes an integer digit and a big integer, and returns the big integer list which is the result of multiplying the big integer with the digit. Once you have implemented the function, you should get the following behavior at the OCaml prompt:

```ocaml
# mulByDigit 9 [9;9;9;9];;
- : int list = [8;9;9;9;1]
```

Now, using the function `mulByDigit` and `bigAdd`, fill in the implementation of `bigMul : int list -> int list -> int list`. Again, you have to fill in implementations for `f`, `base`, `args` only. Once you are done, you should get the following behavior at the prompt:

```ocaml
# bigMul [9;9;9;9] [9;9;9;9];;
- : int list = [9;9;9;8;0;0;0;1]
# bigMul [9;9;9;9;9] [9;9;9;9;9];;
- : int list = [9;9;9;9;8;0;0;0;0;1]
```

**Submission:** Login to autolab and select the Lab 5 assignment. Upload and submit your revised version of misc5.ml.

**NOTE:** Autolab will not detect it if you alter the templates. However, the specs for the assignment say that you must use the templates as they appear in the handout. I will be reviewing code and deducting the points for any problems where the template has been altered!

**Attribution:** This assignment was adapted from Sorin Lerner’s CSE 130 course at UCSD.