

COMP 3958: Lab 2

Submit a file named `lab2.ml` containing your source code. Except otherwise indicated, you may only use functions you have written or functions from the `Stdlib` module. Your file must compile without warnings or errors. If not, you may receive no credit for this lab exercise. Maximum score: 15.

1. Using either `List.fold_left` or `List.fold_right`, implement:

- (a) the `map` function from class with signature

```
val map : ('a -> 'b) -> 'a list -> 'b list
```

- (b) the `dedup` function from lab 1 with signature

```
val dedup: 'a list -> 'a list
```

2. (a) Recall that `filter` has signature

```
val filter : ('a -> bool) -> 'a list -> 'a list
```

and `filter f l` returns all the elements of `l` that satisfy the predicate `f`, preserving the order of the elements in the input list.

Implement from basics a function `filteri` with signature

```
val filteri : (int -> 'a -> bool) -> 'a list -> 'a list
```

that is similar to `filter`, except that for `filteri f l`, the predicate `f` (with type `int -> 'a -> bool`) is applied to the index (starting from 0) as well as the value of each element of `l`. You may use `List.rev`.

- (b) Use `filteri` to implement `filter`.

- (c) Use `filteri` to implement a function named `every` with signature

```
val every : int -> 'a list -> 'a list
```

so that `every n lst` is a list consisting of every `n`th element of `lst`. Requires `n > 0`. For example, `every 3 lst` is a list consisting of every third element of `lst`:

```
every 3 [1; 2; 3; 4; 5; 6; 7; 8; 9; 10] is [3; 6; 9].
```

3. (a) Recall that `List.fold_left` has signature

```
val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a list -> 'acc
```

Implement from basics a function `fold_while` that has signature

```
val fold_while : ('acc -> 'a -> 'acc option) -> 'acc -> 'a list -> 'acc
```

`fold_while` is a “short-circuiting” version of `fold_left`; it is just like `fold_left` except that it can stop early before finish processing the whole list.

Recall that the function passed to `fold_left` is applied to the accumulator and an element of the list and returns the new accumulator. The function passed to `fold_while` is also applied to the accumulator and an element of the list. However, it returns an “optional” accumulator. If the return value is `None`, the processing stops and returns the current accumulator; if the return value is `Some v`, then `v` is the new accumulator. `fold_while` keeps going until either the function returns `None` or until all elements of the given list have been processed.

- (b) Use `fold_while` to implement `fold_left`.

- (c) Given a list `l` of positive integers and a positive integer `n`, we want to keep summing the integers in `l` as long as the total is strictly less than `n`. Use `fold_while` to implement such a function. Name it `sum_while_less_than`. It takes `n` and `l` and returns a pair whose first element is the count of the integers summed and whose second element is their total. For example,

- `sum_while_less_than 20 [6; 5; 5; 3; 4]` returns `(4, 19)`;
- `sum_while_less_than 6 [6; 5; 5; 3; 4]` returns `(0, 0)`.

The function has signature

```
val sum_while_less_than : int -> int list -> int * int
```