T-501-FMAL Programming languages, Practice class 2
Spring 2021

1. (i) Code in F# directly by recursion a function `countOcc : ’a -> ’a list -> int` that counts the number of times a given value occurs in a given list.

```
> countOcc 3 [0;7;4;3;5;7;7;3;8;6;17];;
val it : int = 2
```

Code the same function using `List.fold`.

Code the same function using `List.filter` and `List.length`.

(ii) Using `counOcct`, code a function `occurs : ’a -> ’a list -> bool when’a : equality` that determines if a given value occurs in a given list.

```
> occurs 7 [1;7;4;3;5;7;7;3;8;6;17];;
val it : bool = true
> occurs 15 [1;7;4;3;5;7;7;3;8;6;17];;
val it : bool = false
```

Code the same function using `List.filter`.

Code this function also directly by recursion...

... and using `List.fold`.

... and using `List.exists`.

The definition directly by recursion (if you wrote it right) is more efficient than those using `countOcc` and `List.filter` and also more efficient than by those using `List.fold` and `List.exists`. Why is this?

2. Code a function `sorted : ’a list -> bool when ’a : comparison` that checks whether a list is sorted (non-strictly increasing) directly by recursion.

```
> sorted [0;2;5;5;8;13];;
val it : bool = true
> sorted [3;4;1;6;18;18;20;47];;
val it : bool = false
```

Code a function `pairs : ’a list -> (’a * ’a) list` that extracts from a list all consecutive pairs of elements (this very similar to `groups3` from Practical 1).

```
> pairs [0;2;5;5;8;13];;
val it : (int * int) list = [(0,2);(2,5);(5,5);(5,8);(8,13)]
```

Code `sorted` using `pairs` and `List.fold`.

Code `sorted` using `pairs` and `List.forall`.

3. (i) Code (in any reasonable way you please) a function `removeFirst : ’a -> ’a list -> ’a list when ’a : equality` that removes from a given list the first occurrence of a given value (if it occurs there at all), but keeps all later occurrences.

```
> removeFirst 7 [0;7;4;3;5;7;7;3;8;6;17];;
val it : int list = [0;4;3;5;7;7;3;8;6;17]
```

(ii) Code a function `remove : ’a -> ’a list -> ’a list when ’a : equality` that removes from a given list all occurrences of a given value.

```
> remove 7 [0;7;4;3;5;7;7;3;8;6;17];;
val it : int list = [0;4;3;5;3;8;6;17]
```

(iii) Use `remove` to code a function `nub : ’a list -> ’a list when ’a : equality` that makes a given list duplicate-free by keeping only the first occurrence of each value in the list.

```
> nub [0;7;4;3;5;7;7;3;8;6;17];;
val it : int list = [0;4;3;5;8;6;17]
```

4. (i) Code a function `suffixes : 'a list -> ('a list) list` that lists all suffixes of a given list (in the order you find most convenient).

```
> suffixes [3;1;8;2;17];;
val it : int list list = [[3;1;8;2;17];[1;8;2;17];[8;2;17];[2;17];[17];[]]
```

(ii) Code a function `prefixes : 'a list -> ('a list) list` that lists all suffixes of a given list (in the order you find most convenient).

```
> prefixes [3;1;8;2;17];;
val it : int list list = [[];[3];[3;1];[3;1;8];[3;1;8;2];[3;1;8;2;17]]
```

(iii) Code a function `sublists : 'a list -> ('a list) list` that lists all sublists of a given list where by a sublist I mean a list obtained from the given one by dropping some elements.

```
> sublists [3;1;8];;
val it : int list list =
  [[3;1;8];[1;8];[3;8];[8];[3;1];[1];[3];[]]
```

5. (i) Consider the type of node-labelled binary trees from the lecture.

```
type 'a tree =
    | Lf
    | Br of 'a * 'a tree * 'a tree
```

Code a function `subtreeSums : int tree -> int tree` that replaces each node label with the sum of all node labels below it (including the node itself)

```
> subtreeSums (Br (100,Br (20,Lf,Br (4,Lf,Lf)),Br (30,Br (5,Lf,Lf),Lf)));;
val it : int tree =
               Br (159,Br (24,Lf,Br (4,Lf,Lf)),Br (35,Br (5,Lf,Lf),Lf))
```

(ii) Code a function `pathSums : int tree -> int tree` that replaces each node label with the sum of all node labels on path from the root to that node.

```
> pathSums (Br (100,Br ( 20,Lf,Br (  4,Lf,Lf)),Br ( 30,Br (  5,Lf,Lf),Lf)));;
val it : int tree =
            Br (100,Br (120,Lf,Br (124,Lf,Lf)),Br (130,Br (135,Lf,Lf),Lf))
```