# SC-T-501-FMAL Programming languages, Assignment 1
# Spring 2020
# Due 11 Feb 2020 at 23:59

1. Write a function `withinBounds : int -> int -> int list -> bool` such that `withinBounds min max xs` tests whether all of the elements of `xs` are between `min` and `max` (inclusive).

```
> withinBounds 1 10 [];;
val it : bool = true
> withinBounds 1 10 [1..10];;
val it : bool = true
> withinBounds 1 10 [1; 0; 2];;
val it : bool = false
```

2. (i) Write directly by recursion a function `findSum : int -> int list -> int` that takes a target sum and a list `xs`, and returns the smallest `n` such that the sum of the first `n` elements of the list is the target sum. Your function should return the length if there is no such `n`.

```
> findSum 0 [];;
val it : int = 0
> findSum 2 [1;-1;3;-1];;
val it : int = 4
> findSum 2 [1;-1;3;-1;1;5];;
val it : int = 4
> findSum 5 [1;5;-2;3;-7];;
val it : int = 5
```

   (ii) Reimplement `findSum` using `fold`, by filling in the `...` in the following declaration:

```
let findSum2 sum xs =
    let (_, n) =
        List.fold (fun (r, i) -> fun x -> ...) (sum, 0) xs
    n
```

3. A list `xs : int list` is *well-bracketed* if both of the following are true:

   - For each `n`, the sum of the first `n` elements of `xs` is non-negative.
   - The sum of all of the elements of `xs` is zero.

   Write a function `isBracketed : int list -> bool` that checks whether a list is well-bracketed.

```
> isBracketed [];;
val it : bool = true
> isBracketed [1;-1;1;2;-3];;
val it : bool = true
> isBracketed [1;-2;2;-1];;
val it : bool = false
> isBracketed [1;1;-2;3;-2];;
val it : bool = false
```

4. Consider the following functions:

```
// lookup : 'a -> string -> (string * 'a) list -> 'a
let rec lookup dv (k : string) = function
    | [] -> dv
    | (k', v) :: xs -> if k = k' then v else lookup dv k xs

// update : string -> 'a -> (string * 'a) list -> (string * 'a) list
```

```
let rec update (k : string) v = function
    | [] -> [(k, v)]
    | (k', _) as p :: xs ->
        if k = k' then (k, v) :: xs else p :: update k v xs
```

(i) Implement a function `count : string -> string list -> int` such that `count x xs` is the number of times the string `x` appears in the list `xs`.

```
> count "x" ["y";"z"];;
val it : int = 0
> count "x" ["x";"x";"x"];;
val it : int = 3
> count "x" ["y";"x";"x";"z";"y";"y"];;
val it : int = 2
```

(ii) The following function traverses its input list twice:

```
// modify : 'a -> ('a -> 'a) -> string -> (string * 'a) list -> (string * 'a) list
let modify d f x xs = update x (f (lookup d x xs)) xs
```

Reimplement `modify` so that the list is traversed at most once.

(iii) Using `modify`, implement a function
`ac : (string * int) list -> string list -> (string * int) list`
that adds the number of times each string appears in the second list. Your function should satisfy `lookup 0 x (ac dict xs) = lookup 0 x dict + count x xs` for all `dict, x, xs`.

```
> ac [] ["x";"x";"y";"x";"y"];;
val it : (string * int) list = [("x", 3); ("y", 2)]
> ac [("x", 2);("y",3)] ["x";"y";"z"];;
val it : (string * int) list = [("x", 3); ("y", 4); ("z", 1)]
> ac [("x", 2);("y",3)] ["x";"x";"x"];;
val it : (string * int) list = [("x", 5); ("y", 3)]
```

5. Consider the following function `uf : ('b -> ('a * 'b) option) -> 'b -> 'a list`.

```
let rec uf f x =
    match f x with
    | None -> []
    | Some (a, y) -> a :: uf f y
```

Complete the following definition of `fromOne : int -> int list` by replacing `...` with a non-recursive expression so that
`fromOne n = [1..n]`:

```
let fromOne n = uf (...) 1
```

```
> fromOne 0;;
val it : int list = []
> fromOne 1;;
val it : int list = [1]
> fromOne 5;;
val it : int list = [1; 2; 3; 4; 5]
```

6. The type `'a tree` represents binary trees containing elements of `'a`, and `pos` represents positions in such trees (e.g. `L (R S)` means go left, then go right, then stop).

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

type pos =
    | S                 // stop here
    | L of pos          // go left
    | R of pos          // go right
```

Write a function `deleteSubtree : 'a tree -> pos -> 'a tree` that replaces the subtree at a given position with `Lf` if the position exists in the tree and leaves the tree unchanged otherwise.

```
> let t = Br (1, Br(2, Lf, Lf), Br (3, Br (4, Lf, Lf), Lf));;
val t : int tree = Br (1,Br (2,Lf,Lf),Br (3,Br (4,Lf,Lf),Lf))
> deleteSubtree t S;;
val it : int tree = Lf
> deleteSubtree t (L S);;
val it : int tree = Br (1,Lf,Br (3,Br (4,Lf,Lf),Lf))
> deleteSubtree t (R (L S));;
val it : int tree = Br (1,Br(2,Lf,Lf),Br (3,Lf,Lf))
> deleteSubtree t (R (R (R S)));;
val it : int tree = Br (1,Br(2,Lf,Lf),Br (3,Br (4,Lf,Lf),Lf))
```

7. Consider the following datatype:

```
type fp =
    | Nil
    | IntCons of int    * fp
    | StrCons of string * fp
```

(i) Write a function `fromIntList : int list -> fp` that takes a list of integers and converts it into an element of `fp` by replacing `::` with `IntCons`.

```
> fromIntList [];;
val it : fp = Nil
> fromIntList [1;3;9];;
val it : fp = IntCons (1,IntCons (3,IntCons (9,Nil)))
```

(ii) Write a function `extractInts : fp -> int list` that extracts the integers from the given element of `fp`.

```
> extractInts (fromIntList [1..5]);;
val it : int list = [1; 2; 3; 4; 5]
> extractInts (IntCons (1, StrCons ("x", fromIntList [5..7])));;
val it : int list = [1; 5; 6; 7]
> extractInts (StrCons ("x", Nil));;
val it : int list = []
```

(iii) Write a function `valid : fp -> bool` that returns `true` when there are no two adjacent `int`s in the argument, and no two adjacent `string`s.

```
> valid Nil;;
val it : bool = true
> valid (StrCons("x", Nil));;
val it : bool = true
> valid (IntCons(1, StrCons("x", IntCons(2, StrCons("y", Nil)))));;
val it : bool = true
> valid (StrCons("x", IntCons(1, IntCons(2, Nil))));;
val it : bool = false
> valid (StrCons("x", StrCons("y", Nil)));;
val it : bool = false
```

(iv) Write a function `norm : fp -> fp` that sums adjacent `int`s and concatenates adjacent `string`s in an `fp`, so that `valid (norm l) = true` for all `l`.

```
> norm Nil;;
val it : fp = Nil
> norm (IntCons(1, StrCons ("x", IntCons (3, StrCons("y", Nil)))))
val it : fp = IntCons (1, StrCons ("x",IntCons (3,StrCons ("y",Nil))))
> norm (fromIntList [1..10]);;
val it : fp = IntCons (55,Nil)
> norm (IntCons(1, StrCons ("Hello, ", StrCons ("World!", IntCons(2, Nil)))));;
val it : fp = IntCons (1, StrCons ("Hello, World!",IntCons (2,Nil)))
```