Homework 1
ENE4014 Programming Languages, Spring 2026
due: 4/22(Wed), 23:59

- Submit one file per problem via the submission system in the course website. Make sure that your files are compiled and run without errors. Any file that does not compile or run will receive zero points. You should include all auxiliary or helper functions in the same file.
- Do not use any external libraries. Any code that uses external libraries (including the standard library (`https://ocaml.org/api/index.html`)) will receive zero points.

Exercise 1 Natural numbers are defined inductively as follows:

$$\overline{0} \qquad \frac{n}{n+1}$$

The inductive definition can be defined by the following data type in OCaml:

```
type nat = O | S of nat
```

For example, `O` denotes zero, `S O` denotes 1, and `S (S O)` denotes 2. Write two functions that add and multiply natural numbers:

```
natadd : nat -> nat -> nat
natmul : nat -> nat -> nat
```

For example, the functions should work as follows:

```
# let two = S (S O) ;;
val two : nat = S (S O)
# let three = S (S (S O));;
val three : nat = S (S (S O))
# natadd two three ;;
- : nat = S (S (S (S (S O))))
# natmul two three;;
- : nat = S (S (S (S (S (S O)))))
```

Exercise 2 Write two functions

```
is_even : nat -> bool
leq : nat -> nat -> bool
```

where `is_even` returns true if a given natural number is an even number (otherwise, returns false), and `leq n1 n2` returns true if `n1` $\leq$ `n2` (otherwise, returns false). For example,

```
# is_even two ;; ;;
- : bool = true
# leq two three;;
- : bool = true
# leq three two ;;
- : bool = false
# is_even three ;;
- : bool = false
```

Exercise 3 Write a function

```
is_squared : nat -> bool
```

that returns true if a given natural number is a square number. Zero is not a square number. For example,

```
# is_squared two ;;
- : bool = false
# let nine = natmul three three ;; ;;
val nine : nat = S (S (S (S (S (S (S (S (S O))))))))
# is_squared nine ;; ;;
- : bool = true
# let four = natmul two two ;; ;;
val four : nat = S (S (S (S O)))
# is_squared four ;; ;;
- : bool = true
```

Exercise 4 Lists of natural numbers are defined inductively as follows:

$$\frac{}{\texttt{nil}} \qquad \frac{l}{n \cdot l}\ n \in \mathbb{N}$$

The inductive definition can be defined by the following data type in OCaml:

```
type nat_list = Nil | Cons of nat * nat_list
```

where `nat` refers to the type for natural numbers defined in Exercise 1.

Write a function

```
for_all : (nat -> bool) -> nat_list -> bool
```

such that `for_all f l` returns false if any element of `l` fails to satisfy `f`, and returns true otherwise. For example,

```
# for_all is_even Nil;;
- : bool = true
# let lst1 = Cons (two, Cons(three, Nil)) ;; ;;
val lst1 : nat_list = Cons (S (S O), Cons (S (S (S O)), Nil))
# let lst2 = Cons (two, Cons(four, Nil)) ;; ;;
val lst2 : nat_list = Cons (S (S O), Cons (S (S (S (S O))), Nil))
# for_all is_even lst1 ;; ;;
- : bool = false
# for_all is_even lst2 ;; ;;
- : bool = true
```

Exercise 5 Write a function

```
filter_not : (nat -> bool) -> nat_list -> nat_list
```

such that `filter_not f l` returns the list of those elements in `l` that does NOT satisfy the predicate. For example,

```
# filter_not is_even lst1 ;;
- : nat_list = Cons (S (S (S O)), Nil)
# filter_not (fun n -> leq n (S (S O))) lst2;;
- : nat_list = Cons (S (S (S (S O))), Nil)
```

Exercise 6 Write a function

```
npower:  int -> int -> float
```

that returns $\frac{1}{x^n}$ for two given integers x and $n(\geq 0)$. x^0 is defined to be 1.

Exercise 7 Write a function

```
gcd:  int -> int -> int
```

that returns the greatest common divisor (GCD) of two given non-negative integers. Use the Euclidean algorithm based on the following definition (for two integers n and m $(n \geq m)$):

$$\texttt{gcd}\ n\ m = \begin{cases} n & (m = 0) \\ \texttt{gcd}\ (n - m)\ m & \end{cases}$$

Exercise 8 Write a function

```
min:  int list -> int
```

that returns the minimum value of a given list of integers. If the list is empty, return 0.

Exercise 9 Write a function

```
cartesian:  'a list -> 'b list -> ('a * 'b) list
```

that returns a list of from two lists. That is, for lists A and B, the Cartesian product $A \times B$ is the list of all ordered pairs (a, b) where $a \in A$ and $b \in B$. For example, if $A = [\text{“}a''; \text{“}b''; \text{“}c'']$ and $B = [1; 2; 3]$, $A \times B$ is defined to be

$$[(\text{“}a'', 1); (\text{“}a'', 2); (\text{“}a'', 3); (\text{“}b'', 1); (\text{“}b'', 2); (\text{“}b'', 3); (\text{“}c'', 1); (\text{“}c'', 2); (\text{“}c'', 3)]$$

Binary trees can be defined as follows:

```
type btree = Leaf | Node of int * btree * btree
```

The number in the `Node` constructor is called the *key* of the node.

Exercise 10 Write a function

```
count_leaves :  btree -> int
```

that takes a binary tree and returns the number of all leaves in the tree. For example,

```
# let t =  Node (2, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf)) ;;
val t : btree = Node (2, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf))
# count_leaves t ;;
- : int = 4
```

Exercise 11 Write a function

```
count_oddnode :  btree -> int
```

that takes a binary tree and returns the number of odd keys in the tree. For example,

```
# let t =  Node (1, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf)) ;;
val t : btree = Node (2, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf))
# count_oddnode t ;;
- : int = 2
```

Exercise 12 Write a function

```
insert_btree :  int -> btree -> btree
```

that takes an integer and a binary search tree and returns a new binary search tree with the integer properly inserted in the tree. A binary search tree (BST) is a tree where the key of each node is greater than all keys in its left subtree and less than all keys in its right subtree. For example,

```
# let t =  Node (2, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf)) ;;
val t : btree = Node (2, Node (2, Leaf, Leaf), Node (3, Leaf, Leaf))
# insert_btree 1 t ;;
- : btree = Node (2, Node (2, Node (1, Leaf, Leaf), Leaf), Node (3, Leaf, Leaf))
```

Exercise 13 Write a function

```
double_tree : btree -> btree
```

that takes a binary tree and produces another binary tree like the original, but with all the integers in the leaves doubled. For example,

```
# double_tree t1 ;;
- : btree = Node (2, Leaf, Leaf)
# double_tree t2;;
- : btree = Node (2, Node (4, Leaf, Leaf), Node (6, Leaf, Leaf))
```

Exercise 14 Write a function `tree_path`

```
tree_path : btree -> int -> path
```

that takes an integer `n` and a binary tree `t`, and returns a *path*, which is a list of `Lefts` and `Rights` showing how to find the node containing `n` starting from the root.

- If `n` is found at the root, the path is the empty list.
- If the tree does not contain `n`, `NoPath` should be returned.
- You can assume all integers in a tree are unique.

The paths are inductively defined as follows:

```
type path = NoPath | Path of direction list
and direction = Left | Right
```

The function should work as follows:

```
# let t3 = Node (1, Node (2, Node (3, Leaf, Node (4, Leaf, Leaf)), Leaf), Leaf) ;;
val t3 : btree =
  Node (1, Node (2, Node (3, Leaf, Node (4, Leaf, Leaf)), Leaf), Leaf)
# tree_path t3 5 ;;
- : path = NoPath
# tree_path t3 1 ;;
- : path = Path []
# tree_path t3 4 ;;
- : path = Path [Left; Left; Right]
```

Exercise 15 Write a function

```
duplicate:  'a list -> 'a list
```

that duplicates the elements of a list. For example,

```
duplicate [1; 2; 3] = [1; 1; 2; 2; 3; 3].
```

Exercise 16 Write a function

```
replicate:  'a list -> int -> 'a list
```

that replicates the elements of a list a given number $n(\geq 0)$ of times. If n is 0, the function should return an empty list. For example,

```
replicate [1; 2; 3] 3 = [1; 1; 1; 2; 2; 2; 3; 3; 3].
```

Exercise 17 Write a function

```
deduplicate:  'a list -> 'a list
```

that takes a list and returns a list with all duplicates removed. The order of the elements in the result should be the same as the order in the original list. For example,

```
deduplicate [1; 1; 2; 2; 3; 3; 2; 2] = [1; 2; 3].
```

Exercise 18 Write a function

```
lall:  'a list -> ('a -> bool) -> bool
```

such that

$$\texttt{lall}\ l\ p = \begin{cases} \texttt{true} & \text{(if } p \text{ holds for all elements of } l\text{)} \\ \texttt{false} & \text{(otherwise)} \end{cases}$$

For example,

```
lall [1; 2; 3] (fun x -> x > 0) = true
```

and

```
lall [1; 2; 3] (fun x -> x > 1) = false.
```

Exercise 19 Write a function

```
lany:  'a list -> ('a -> bool) -> bool
```

such that

$$\texttt{lany}\ l\ p = \begin{cases} \texttt{true} & \text{(if } p \text{ holds for at least one element of } l\text{)} \\ \texttt{false} & \text{(otherwise)} \end{cases}$$

For example,

```
lany [1; 2; 3] (fun x -> x mod 2 = 0) = true
```

and

```
lany [1; 2; 3] (fun x -> x < 0) = false.
```

Exercise 20 Write a function

```
powerset:  'a list -> 'a list list
```

such that `powerset` l returns the list of all subsets of l. For example, if $l = [1; 2; 3]$, then `powerset` l is defined to be

$$[[]; [1]; [2]; [3]; [1; 2]; [1; 3]; [2; 3]; [1; 2; 3]].$$

You don't have to consider the order of the elements in the result. For example, both $[[2; 1]; [1]; [2]; []]$ and $[[1]; [1; 2]; [2]; []]$ are correct answers for `powerset` $[1; 2]$.