(* Wooyoung Jung 114744214- CSE216_11h *)

(* Exercise 1. (points = 25)
Write a function drop: int -> 'a list-> 'a list such that drop n returns all but the first n elements of lst.
If lst has fewer than n elements, return the empty list. Here, n can be any integer including negative numbers.
*)

```
let rec drop n (lst: 'a list) : 'a list =
    if n <= 0 then lst else
      match lst with
        [] -> []
        | h::t -> if n = 1 then t else drop (n-1) t
```

(* Exercise 2. (points = 25)
Suppose a weighted undirected graph is represented as a list of edges. Each edge is a triple of the type string * string * int,
where the two nodes are represented by edges, and the weight is an integer.

1. Write a type edge to represent an edge
2. Write a type graph to represented a weighted undirected graph
3. Write an OCaml function of type graph -> edge option to identify the minimum weight edge in this graph.
Solve this problem using recursion and pattern matching.
*)

```
type edge = (string * string * int)
type graph = edge list

let rec min_weight (gr: graph) : edge option =
  match gr with
    [] -> None
    | [n] -> Some n
    | e1::e2::t ->
        let (_, _, w1) = e1 in
        let (_, _, w2) = e2 in

        if w1 <= w2 then min_weight (e1::t) (* if two edges are equal, return the
first occurrence *)
        else min_weight (e2::t)
```

(* Exercise 3. (points = 25)

```
  Binary trees can be defined as follows:
      type btree = Empty | Node of int * btree * btree

  For example, the following t1 and t2
      let t1 = Node(1, Empty, Empty)
      let t2 = Node(1, Node (2, Node (3, Empty, Empty), Empty), Node (4, Empty,
Empty))
  are binary trees.

  Write a function mirror: btree -> btree that exchanges the left and right
subtrees all the way down.
  For example,
      mirror t1 = Node (1, Empty, Empty)
      mirror t2 = Node (1, Node (4, Empty, Empty), Node (2, Empty, Node (3,
Empty, Empty)))
*)

type btree = Empty | Node of int * btree * btree

let rec mirror (tree: btree) : btree =
  match tree with
    Empty -> Empty
    | Node(a, left, right) -> Node(a, mirror right, mirror left)

let t1 = Node(1, Empty, Empty)
let t2 = Node(1, Node (2, Node (3, Empty, Empty), Empty), Node (4, Empty,
Empty))

let _ = mirror t1
let _ = mirror t2

(* Exercise 4. (points = 25)
  Natural numbers can be defined as follows:
    type nat = ZERO | SUCC of nat

  For instance, SUCC ZERO denotes 1 and SUCC (SUCC ZERO) denotes 2.

  Write three functions that add, multiply, and exponentiate natural numbers:
    natadd : nat -> nat -> nat
    natmul : nat -> nat -> nat
    natexp : nat -> nat -> nat

  For example,
  # let two = SUCC (SUCC ZERO);;
  val two : nat = SUCC (SUCC ZERO)
  # let three = SUCC (SUCC (SUCC ZERO));;
```

```
  val three : nat = SUCC (SUCC (SUCC ZERO))
  # natadd two three;;
  - : nat = SUCC (SUCC (SUCC (SUCC (SUCC ZERO))))
  # natmul two three;;
  - : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))
  # natexp two three;;
  - : nat = SUCC (SUCC (SUCC (SUCC (SUCC (SUCC (SUCC (SUCC ZERO)))))))
*)

type nat = ZERO | SUCC of nat

(* TODO: implement natmul & natexp *)
let rec natadd (x: nat) (y: nat) : nat =
  match x with
  ZERO -> y
  | SUCC x' -> SUCC (natadd x' y)

let rec natmul (x: nat) (y: nat) : nat =
  match y with
    ZERO -> ZERO
    | SUCC ZERO -> x
    | SUCC y' -> natadd x (natmul x y')

let rec natexp (x: nat) (n: nat) : nat =
    match n with
    ZERO -> SUCC ZERO
    | SUCC ZERO -> x
    | SUCC n' -> natmul x (natexp x n')

let two = SUCC (SUCC ZERO)
let three = SUCC (SUCC (SUCC ZERO))
let four = SUCC(SUCC(SUCC(SUCC ZERO)))

let _ = natadd two three
let _ = natmul two three
let _ = natexp two three

let _ = natmul three two
let _ = natexp three two
```