Guideline

Due Date: Thursday, 2023-11-16, by 23:59.

Upload your answers as a singular PDF to Brightspace. **We do not use the .ml file this time, but you might still make sure your code compiles before copying here.**

Typewriting is preferred.

Multiple submissions are possible before the due time; the last submission will be graded.

Exercise 1. (points = 25)

Write a function drop : int -> 'a list -> 'a list such that drop n lst 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 number.

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 strings, and the weight is an integer.

1. Write a type `edge` to represent an edge
2. Write a type `graph` to represent 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.

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 ways down. For example,

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

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 expoentiate 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)))))))
```