Lab 11

Problem 1

- Node

```
type 'a node =
    | Null
    | Node of ('a node  * 'a * 'a node)
;;
```

- Insert

```
let rec insert elem root = match root with
    | Null -> Node (Null, elem, Null)
    | Node (left, x, right) ->
        if elem < x then
            Node (insert elem left, x, right)
        else if elem > x then
            Node (left, x, insert elem right)
        else
            root
;;
```

- Search

```
let rec search elem root = match root with
    | Null -> false
    | Node (left, x, right) ->
        if elem < x then
            search elem left
        else if elem > x then
            search elem right
        else
            true
;;
```

- Preorder

```
let rec preorder root = match root with
    | Null -> []
    | Node (left, x, right) ->
        [x] @ preorder left @ preorder right
;;
```

- Inorder

```
let rec inorder root = match root with
    | Null -> []
    | Node (left, x, right) ->
        inorder left @ [x] @ inorder right
;;
```

- Postorder

```
let rec postorder root = match root with
    | Null -> []
    | Node (left, x, right) ->
        postorder left @ postorder right @ [x]
;;
```

- Test Tree:

```
let root = insert 5 Null;;
let root = insert 2 root;;
let root = insert 8 root;;
let root = insert 3 root;;
let root = insert 1 root;;
let root = insert 6 root;;
let root = insert 7 root;;
let root = insert 9 root;;
let root = insert 4 root;;
```

- Test Functions:

```
Printf.printf "Search (3): %b\n" (search 3 root);;
Printf.printf "Search (7): %b\n" (search 7 root);;
Printf.printf "Search (10): %b\n" (search 10 root);;
(* Search (3): true *)
(* Search (7): true *)
(* Search (10): false *)

Printf.printf "Preorder: ";;
List.iter (Printf.printf "%d ") (preorder root);
Printf.printf "\nInorder: ";;
List.iter (Printf.printf "%d ") (inorder root);
Printf.printf "\nPostorder ";;
List.iter (Printf.printf "%d ") (postorder root);
(* Preorder: 5 2 1 3 4 6 7 8 9 *)
(* Inorder: 1 2 3 4 5 6 7 8 9 *)
(* Postorder 1 2 3 4 6 7 8 9 5 *)
```

Problem 2

```
- Expression
type expression =
    | Value of float
    | Add of (expression * expression)
    | Sub of (expression * expression)
    | Mul of (expression * expression)
    | Div of (expression * expression)
    | Fn1 of ((float -> float) * expression)
    | Fn2 of ((float -> float -> float) * expression *
expression)
;;

- Evaluate
let rec evaluate expr = match expr with
    | Value x -> x
    | Add (x, y) -> evaluate x +. evaluate y
    | Sub (x, y) -> evaluate x -. evaluate y
    | Mul (x, y) -> evaluate x *. evaluate y
    | Div (x, y) -> evaluate x /. evaluate y
    | Fn1 (f, x) -> f (evaluate x)
    | Fn2 (f, x, y) -> f (evaluate x) (evaluate y)
;;
```

- Test Expression:

```
let eq =
Add(
    Sub(
        Div(
            Value(6.0),
            Value(2.0)
        ),
        Fn2(
            Float.pow,
            Value(2.0),
            Value(3.0)
        )
    ),
    Fn1(
        Float.cos,
        Mul(
            Value(2.0),
            Fn1(
                Float.asin,
                Value(1.0)
            )
        )
    )
);;
```

- Test Result:

```
Printf.printf "Expression = %.2f\n" (evaluate eq);;
(* Expression = -6.00 *)
```