Important notes about grading:

- **Compiler errors:** In cases where programs cannot be successfully compiled, a grade of zero will be assigned. If you encounter difficulties in compiling your assignment, we encourage you to seek assistance by asking questions on Piazza, attending recitation sessions or consulting during office hours.
- You may rename function arguments, but do not modify function names. Doing so will cause you to fail the tests.
- You must not use any mutation operations of OCaml for any of these questions: no arrays, for- or while-loops, references, etc.
- **You can always add a helper function for any of the functions we ask you to implement, and the helper function can also be recursive**.

Compile and run your code:

After downloading and unzipping the file for assignment3, in your terminal, `cd` into `assignment3`. You should write your code in `assignment3.ml`.

Use `ocamlc -o test ast.ml expressionLibrary.ml assignment3.ml` to compile the code. Then, use `./test` to excute it.

About library functions:

You can use any library functions for this assignment.

Submission:

Please submit `assignment3.ml` to Canvas.

Datatypes:

In OCaml, you can define a tree data structure using a datatype:

```
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

For problem 1 & 2, we will work with binary search trees and define a function for inserting elements into a binary search tree as follows::

```
let rec insert tree x =
  match tree with
  | Leaf -> Node(Leaf, x, Leaf)
  | Node(l, y, r) ->
     if x = y then tree
     else if x < y then Node(insert l x, y, r)
     else Node(l, y, insert r x)
```

A binary search tree can be constructed from a list:

```
let construct l =
  List.fold_left (fun acc x -> insert acc x) Leaf l
```

Problem 1

We have seen the benefits of the 'fold' function for list data structures. In a similar fashion, write a function

```
fold_inorder : ('a -> 'b -> 'a) -> 'a -> 'b tree -> 'a
```

that does an inorder fold of the tree. The function should traverse the left subtree, visit the root, and then traverse the right subtree.

In [ ]:
```
let rec fold_inorder f acc t =
  (* YOUR CODE HERE *)
```

In [ ]:
```
assert (fold_inorder (fun acc x -> acc @ [x]) [] (Node (Node (Leaf,1,Leaf), 2, Node (Leaf,3,Leaf))) = [1;2;3]);
assert (fold_inorder (fun acc x -> acc + x) 0 (Node (Node (Leaf,1,Leaf), 2, Node (Leaf,3,Leaf))) = 6)
```

Problem 2

Given a binary search tree `t`, return the tree after removing an element `x` from `t`.

```
remove : 'a -> 'a tree -> 'a tree
```

Note: During the lecture, an implementation for remove was discussed that you can refer to. In that implementation, if the node containing the value to be deleted has two children, it follows these steps: (1) finds an inorder predecessor of the node (refer to problem 1), (2) uses the value of the inorder predecessor as the new value of the node, and (3) removes the inorder predecessor. However, for your solution to Problem 10, please ensure you use the inorder successor instead.

Example:

```
       50                        60
     /    \         remove 50     /    \
   40      70       --------->   40     70
          /  \                            \
         60   80                           80
```

In [ ]:
```
let rec remove x t =
    (* YOUR CODE HERE *)
```

In [ ]:
```
assert (remove 50 (Node (Node (Leaf, 40, Leaf), 50, Node (Node (Leaf, 60, Leaf), 70, Node (Leaf, 80, Leaf))))
                      = (Node (Node (Leaf, 40, Leaf), 60, Node (Leaf, 70, Node (Leaf, 80, Leaf)))))
```

A Language for Symbolic Differentiation

The objective of this part is to build a language that can differentiate and evaluate symbolically represented mathematical expressions that are functions of a single variable. Symbolic expressions consist of numbers, variables, and a subset of the standard math functions (plus, minus, and times). Nowadays, symbolic differentiation of algebraic expressions is a task that can be conveniently accomplished on modern mathematical packages, such as Mathematica and Maple.

To get you started, we have provided the datatype that defines the abstract syntax tree for such expressions in `ast.ml`:

```
(* abstract syntax tree *)

(* Binary operators. *)
type binop = Add | Sub | Mul

type expression =
  | Num of float
  | Var
  | Binop of binop * expression * expression
```

`Var` represents an occurrence of the single variable `x`. `Binop(Add, Var, Num 3.0)` represents the expression that adds `3.0` to `x`. The operators `Sub` and `Mul` refer to subtraction and multiplication, respectively. More complicated mathematical expressions involving addition, subtraction, multiplication, constants and the variable `x` can be constructed using combinations of the constructors in the above datatype definition. For example, `2.0*x + (x*x - 3.0)` can be represented as:

```
Binop(Add, Binop(Mul, Num(2.0), Var), Binop(Sub, Binop(Mul,Var,Var), Num(3.0))))
```

Each such expression represents a tree where nodes are the constructors and the children of each node are the specific operator to use and the arguments of that constructor. Such a tree is called an abstract syntax tree (or AST for short).

Provided Infrastructure

We have provided some functions to create and manipulate expression values. These functions can be helpful for you to debug your code. They are defined in `expressionLibrary.ml`.

- `parse: string -> expression` : translates a string in infix form (such as `x*x - 3.0*x + 2.5`) into an expression (treating `x` as the variable). The parse function parses according to the standard order of operations - so `5+x*8` will be read as `5+(x*8)`.
- `to_string: expression -> string` : prints expressions in a readable form, using infix notation. This function adds parentheses around every binary operation so that the output is completely unambiguous.
- `to_string_wo_paren: expression -> string` : prints expressions in a readable form, using infix notation. This function does not add any parentheses so it can only be used for expressions in standard forms (see the definition below).
- `make_exp: int -> expression` : takes in a length `l` and returns a randomly generated expression of length at most `2l`.
- `rand_exp_str: int -> string` : takes in a length `l` and returns a string representation of length at most `2l`.

In [ ]:
```
let _ =
      (* The following code make an expression from a string
         "5*x*x*x + 4*x*x + 3*x + 2 + 1", and then convert the
         expression back to a string, finally it prints out the
         converted string
         *)
      let e = parse ("5*x*x*x + 4*x*x + 3*x + 2 + 1") in
      let s = to_string e in
      print_string (s^"\n")

let _ =
      (* The following code make a random expression from a string
         and then convert the expression back to a string
         finally it prints out the converted string
         *)
      let e = make_exp 10 in
      let s = to_string e in
      print_string (s^"\n")
```

Problem 3: Expression Evaluation

Your first take is to write a function that will evaluate (i.e., interpret) an expression.

Write a function

```
evaluate: expression -> float -> float
```

such that given an expression `e` and a floating point value `v` for the single variable `x`, `evaluate e v` produces the floating point result of evaluating expression `e` when `x` is `v`. For example,

```
evaluate (parse "x*x + 3.0") 2.0 = 7.0
```

In [ ]:
```
let rec evaluate (e:expression) (x:float) : float =
    (* YOUR CODE HERE *)
```

In [ ]:
```
assert (evaluate (parse "x*x + 3.0") 2.0 = 7.0);
```

Problem 4: Derivatives

Write a function

```
derivative: expression -> expression
```

such that `derivative e` takes an expression `e` as its argument and returns an expression `e'` representing the derivative of the expression `e` with respect to the single variable `x`. This process is referred to as symbolic differentiation.

If you don't remember your calculus, here's the necessary crib sheet, some formulae for computing derivatives that you will use:

```
derivative(f + g)(x) = derivative(f)(x) + derivative(g)(x)
derivative(f - g)(x) = derivative(f)(x) - derivative(g)(x)
derivative(f * g)(x) = derivative(f)(x) * g(x) + f(x) * derivative(g)(x)
```

In addition, the derivative of any constant value is `0` and the derivative of the single variable `x` is 1.

The result of your function must be correct, but need not be expressed in the simplest form. Take advantage of this in order to keep the code in this part as short as possible.

In [ ]:
```
let rec derivative (e:expression) : expression =
    (* YOUR CODE HERE *)
```

In [ ]:
```
assert (evaluate (derivative (parse "x*x + 3.0")) 2.0 = 4.0);
```

Problem 5: Zero Finding

One application of the derivative of a function is to find zeros of a function. One way to do so is [Newton's method].

Write a function

```
find_zero:expression -> float -> float -> int -> float option
```

to implement the Newton's method.

The function `find_zero e g epsilon lim` should take an expression `e`, a guess for the zero `x`, a precision requirement `epsilon`, and a limit to the number of iterations `lim`. If a guess x_n (to include the starting guess x_0) is sufficient to find a zero within the required precision, it should immediately return `Some` x_n. Otherwise, so long as the limit has not been reached, it should produce a new guess x_{n+1} and try again. It should return `None` if no zero was found within the desired precision `epsilon` by the time the limit `lim` was reached.

Note: If the expression that find_zero is operating on is $f(x)$ and the precision is `epsilon`, we are asking you to find a value x such that $|f(x)|$ < `epsilon`. That is, the value that the expression evaluates to at x is "within `epsilon`" of 0.

We are not asking you to find an x such that $|x - x_0|$ < `epsilon` for some x_0 for which $f(x_0) = 0$.

In [ ]:
```
let find_zero (e:expression) (xn:float) (epsilon:float) (lim:int)
  : float option =
    (* YOUR CODE HERE *)
```

In [ ]:
```
let e = (parse "2*x*x - x*x*x - 2") in
let g, epsilon, lim = 3.0, 1e-3, 50 in
let x = find_zero e g epsilon lim in
match x with
| None -> assert false
| Some x ->
  let eval_result = evaluate e x in
  assert (0. -. epsilon < eval_result && eval_result < epsilon)
```

Problem 6: Simplification

Write a function

```
simplify: expression -> expression
```

such that given an expression `e`, `simplify e` reduces it to its simplest polynomial form by

- combining like terms by adding or subtracting their coefficients.
- simplifing expressions involving multiplication, distributing the terms if necessary.
- Arranging the terms in descending order of degrees (from highest to lowest).
- Ensuring that the polynomial is in the form $ax^n + bx^{n-1} + ... + cx + d$ where a, b, c and d are constants. If the coefficient of any monomial is 0, that term is omitted from the final expression.

Examples:

- $3 * x * x + 2 * x - 5 + 4 * x * x - 7 * x$ should be reduced to $7 * x * x - 5 * x - 5$.
- $(x - 1) * (x * (x - 5)$ should be reduced to $1 * x * x * x - 6 * x * x + 5 * x$.
- $x - x$ should be reduced to 0.

In [ ]:
```
let simplify (e:expression) : expression =
    (* YOUR CODE HERE *)
```

In [ ]:
```
assert (to_string_wo_paren (simplify (parse "3*x*x + 2*x - 5 + 4*x*x - 7*x")) = "7.*x*x+-5.*x+-5.");
assert (to_string_wo_paren (simplify (parse "(x-1)*x*(x-5)")) = "1.*x*x*x+-6.*x*x+5.*x");
assert (to_string_wo_paren (simplify (parse "x - x")) = "0.");
assert (to_string_wo_paren (simplify (parse "x + x + 0")) = "2.*x");
assert (to_string_wo_paren (simplify (parse "0")) = "0.")
```

Problem 7: Forward-mode Automatic Differentiation (optional bonous question, 1 point)

There is an optional problem `evaluate2` for implementing forward-mode automatic differentiation. Automatic Differentiation (AD) is really important now in machine learning, as an efficient algorithm for training neural nets. Specifically, the task is to write a function

```
evaluate2: expression -> float -> float * float
```

such that `evaluate2 e x` computes both `e(x)` and the first derivative `e'(x)`, without ever explicitly calculating (`derivative e`). Like `evaluate`, do it by case analysis on the syntax-tree of `e`.

The implementation is easy. The hard part is figuring out what is forward-mode automatic differentiation. It's explained in section 3.1 (and Table 2) of [Automatic Differentiation in Machine Learning: A Survey].

In [ ]:
```
let rec evaluate2 (e: expression) (x: float) : float * float =
    (* YOUR CODE HERE *)
```

In [ ]:
```
assert (evaluate2 (parse "x*x + 3") 2.0 = (7.0, 4.0));
```