Important notes about grading:

1. **Compiler errors:** All code you submit must compile. Programs that do not compile will probably receive an automatic zero. If you run out of time, it is better to comment out the parts that do not compile, than hand in a more complete file that does not compile.
2. **Late assignments:** Please carefully review the course website's policy on late assignments, as all assignments handed in after the deadline will be considered late. Submitting the incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

How to start:

1. Download the source code here.
2. You will need to implement your code in `assignment2/src/lambda.ml`. We give detailed instructions on the functions that you need to implement below (Problem 1-9).
3. The test-cases (also given below) are provided in `assignment2/src/assignment2.ml`.
4. To compile and run your implementation, we provide two options:
   Option (1): Using `Dune`: type `dune exec src/assignment2.exe` in the command-line window under `assignment2` directory.
   Option (2): Without using `Dune`: First, to compile the code, type `make` in the command-line window under `assignment2` directory. Second, to run the code, type `./assignment2.byte` in the command-line window under `assignment2` directory.

PseudoCode:

Try to solve the problems by yourself first. If you get stuck, please refer to the pseudocode of some of the problems here.

Lambda Calculus Interpreter

In this assignment, you will implement lambda calculus interpreters that use different reduction strategies. The abstract syntax tree (AST) for lambda expressions is the one that we have seen in class:

```
type expr =
  | Var of string
  | Lam of string * expr
  | App of expr * expr
```

You are provided a parser function `parse_string` that converts a string to this AST and a printer function `string_of_expr` that converts the AST to string. For example,

In [ ]:

```
open Syntax
let parse_string = Lambda_parse.parse_string
let string_of_expr = string_of_expr

let _ = parse_string "(\\x.x) (\\y.y)"
let _ = string_of_expr (App (Var "x",Lam("y",App(Var "y", Var "x"))))
```

Here `\\x.x` denotes a lambda abstraction `λx. x`. Similarly `\\y.y` denotes `λy.y`. We use `\\` to represent `λ` in program text.

You will need some helper functions to operate over sets. Since we have not studied set data structure in OCaml. We will use lists instead and implement set functionality on top of lists.

Problem 1

Implement a function using `List.fold_left` or `List.fold_right`

```
mem : 'a -> 'a list -> bool
```

`mem e l` returns `true` if the element `e` is present in the list. Otherwise, it returns false.

In [ ]:

```
let mem e l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (mem "b" ["a";"b";"c"] = true);
assert (mem "x" ["a";"b";"c"] = false)
```

Problem 2

Implement a function using `List.fold_left` or `List.fold_right`

```
remove : 'a -> 'a list -> 'a list
```

`remove e l` returns a list `l'` with all the element in `l` except `e`. `remove` also preserves the order of the elements not removed. If `e` is not present in `l`, then return `l`.

In [ ]:

```
let remove e l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (remove "b" ["a";"b";"c"] = ["a";"c"]);
assert (remove "x" ["a";"b";"c"] = ["a";"b";"c"])
```

Problem 3

Implement a function

```
union : string list -> string list -> string list
```

`union l1 l2` performs set union of elements in `l1` and `l2`. The elements in the result list `l` must be lexicographically sorted. Hint: You may want to use the functions `List.sort` and `remove_stutter` from assignment 1 to implement union. Here is an example of using `List.sort`.

In [ ]:

```
assert (List.sort String.compare ["x";"a";"b";"m"] = ["a";"b";"m";"x"])
```

In [ ]:

```
let union l1 l2 =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (union ["a"; "c"; "b"] ["d"; "b"; "x"; "a"] = ["a"; "b"; "c"; "d"; "x"])
```

Problem 4

Implement a function

```
add : 'a -> 'a list -> 'a list
```

`add e l` does a set addition of element `e` to list `l` and returns a list. The resultant list is sorted. Hint: use one of the above functions to implement `add`.

In [ ]:

```
let add e l =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (add "b" ["a";"c"] = ["a";"b";"c"]);
assert (add "a" ["c"; "a"] = ["a";"c"])
```

Substitution

At the heart of reducing lambda expressions is substitution. Recall from the lecture that substitution requires us to generate fresh variable names that is different from every other name used in the current context. We will use the following helper function `fresh` to generate fresh names.

In [ ]:

```
let r = ref 0

let fresh s =
  let v = !r in
  r := !r + 1;
  s ^ (string_of_int v)
```

It uses mutability features of OCaml which we will study in later lectures. You can use the `fresh` function as follows:

In [ ]:

```
let a = fresh "a"
let b = fresh "b"
```

Problem 5

Implement a function

```
free_variables: expr -> string list
```

that returns the free variables in the given lambda term. Hint: Recall that in this assignment we use lists to represent sets. We do not have duplicated elements in a set. Use the `mem`, `remove`, `union` and `add` helper functions that you have implemented in Problem 1-4 to complete Problem 5-9.

In [ ]:

```
let rec free_variables e =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (free_variables (parse_string "\\x.x") = []);
assert (free_variables (parse_string "\\x.y") = ["y"]);
assert (free_variables (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))") = []);
assert (free_variables (parse_string "((\\x.y) (\\z.a)) (\\x.y)") = ["a";"y"])
```

Problem 6

Implement the function

```
substitute : expr -> string -> expr -> expr
```

where `substitute e x v` does `e[v/x]`. For renaming `x` in `Lam(x,y)` with a fresh name, use `Lam (fresh x, ...)`.

In [ ]:

```
let rec substitute expr a b =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
assert (alpha_equiv
          (substitute (parse_string "\\y.x") "x" (parse_string "\\z.z w"))
          (parse_string "λy.λz.z w"));
assert (alpha_equiv
          (substitute (parse_string "\\x.x") "x" (parse_string "y"))
          (parse_string "λx.x"));
assert (alpha_equiv
          (substitute (parse_string "\\x.y") "y" (parse_string "x"))
          (parse_string "λx0.x"))
```

Problem 7

Implement a single step of the call-by-value reduction. Implement the function

```
reduce_cbv : expr -> expr * bool
```

which does a single step of the call-by-value reduction. Recall that call-by-value reduction is determinisitic. Hence, if reduction is possible, then a single rule applies. `reduce e` returns `(e', true)` if reduction is possible and `e'` is the new expression. `reduce e` returns `(e, false)` if reduction is not possible.

In [ ]:

```
let rec reduce_cbv e =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
let expr, reduced = reduce_cbv (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λx.x) (λz.(λx.x) z)"));

let expr, reduced = reduce_cbv (parse_string "(λx.x) (λz.(λx.x) z)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "λz.(λx.x) z"));

let expr, reduced = reduce_cbv (parse_string "λz.(λx.x) z") in
assert (reduced = false &&
        alpha_equiv expr (parse_string "λz.(λx.x) z"));

let expr, reduced = reduce_cbv (parse_string "(λx.y) ((λx.x x) (λx.x x))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λx.y) ((λx.x x) (λx.x x))"));

let expr, reduced = reduce_cbv (parse_string "x y z") in
assert (reduced = false &&
        alpha_equiv expr (parse_string "x y z"))
```

Problem 8

Implement a single step of the call-by-name reduction. Implement the function

```
reduce_cbn : expr -> expr * bool
```

The rest of the instructions are same as `reduce_cbv`

The rest of the instructions are same as `reduce_cbv`.

In [ ]:

```
let rec reduce_cbn e =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
let expr, reduced = reduce_cbn (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λx.x) (λz.(λx.x) z)"));

let expr, reduced = reduce_cbn (parse_string "(λx.x) (λz.(λx.x) z)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "λz.(λx.x) z"));

let expr, reduced = reduce_cbn (parse_string "λz.(λx.x) z") in
assert (reduced = false &&
        alpha_equiv expr (parse_string "λz.(λx.x) z"));

let expr, reduced = reduce_cbn (parse_string "(λx.y) ((λx.x x) (λx.x x))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "y"));

let expr, reduced = reduce_cbn (parse_string "(\\x.x x) ((\\z.z) y)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λz.z) y ((λz.z) y)"));

; let expr, reduced = reduce_cbn (parse_string "x y z") in
assert (reduced = false &&
        alpha_equiv expr (parse_string "x y z"))
```

Problem 9

Implement a single step of the normal order reduction. Implement the function

```
reduce_normal : expr -> expr * bool
```

The rest of the instructions are same as `reduce_cbv`.

In [ ]:

```
let rec reduce_normal e =
  (* YOUR CODE HERE *)
   raise (Failure "Not implemented")
```

In [ ]:

```
let expr, reduced = reduce_normal (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λx.x) (λz.(λx.x) z)"));

let expr, reduced = reduce_normal (parse_string "(λx.x) (λz.(λx.x) z)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "λz.(λx.x) z"));

let expr, reduced = reduce_normal (parse_string "λz.(λx.x) z") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "λz. z"));

let expr, reduced = reduce_normal (parse_string "(λx.y) ((λx.x x) (λx.x x))") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "y"));

let expr, reduced = reduce_normal (parse_string "(\\x.x x) ((\\z.z) y)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λz.z) y ((λz.z) y)"));

let expr, reduced = reduce_normal (parse_string "f (\\x.x x) ((\\z.z) y)") in
assert (reduced = true &&
```

```
        alpha_equiv expr (parse_string "(λx.x x) y"));

let expr, reduced = reduce_normal (parse_string "(\\x.(\\z.z) y) (\\x.x x)") in
assert (reduced = true &&
        alpha_equiv expr (parse_string "(λz.z) y"));
```

Full Reduction and Code Debugging

The 'reduce_cbv', 'reduce_cbn' and 'reduce_normal' functions just implement a single-step evaluation. The following functions recursively apply a reduction function to evaluate a lambda term multiple times until there is no reduction to perform. These functions can also print out intermediate steps of an evaluation, useful for debugging your implementation.

In [ ]:

```
let rec eval log depth reduce expr =
  if depth = 0 then failwith "non-termination?"
  else begin
    let expr', reduced = reduce expr in
    if not reduced then expr else begin
      if log then print_endline ("= " ^ (string_of_expr expr'));
      eval log (depth-1) reduce expr'
    end
  end

let eval_cbv = eval true 1000 reduce_cbv
let eval_cbn = eval true 1000 reduce_cbn
let eval_normal = eval true 1000 reduce_normal
```

Here are some examples of how to use these functions:

In [ ]:

```
let _ = eval_cbv (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))")
let _ = print_endline ""

let _ = eval_cbn (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))")
let _ = print_endline ""

let _ = eval_normal (parse_string "(\\x.x) ((\\x.x) (\\z.(\\x.x) z))")
let _ = print_endline ""
```

You can also write test-cases with these functions.

In [ ]:

```
let zero = parse_string "\\f.\\x. x" in
let one = parse_string "\\f.\\x. f x" in
let two = parse_string "\\f.\\x. f (f x)" in
let three = parse_string "\\f.\\x. f (f (f x))" in

let plus = parse_string "λm. λn. λs. λz. m s (n s z)" in
let mult = parse_string "λm. λn. λs. λz. m (n s) z" in

assert (alpha_equiv (eval_normal (App (App (plus, one), two))) three);
print_endline "";
assert (alpha_equiv (eval_normal (App (App (mult, one), three))) three);
print_endline "";
assert (alpha_equiv (eval_normal (App (App (mult, zero), three))) zero)
```