Important notes about grading:

1. **Compiler errors:** Programs that cannot be compiled will receive an automatic zero. If you are having trouble getting your assignment to compile, please visit recitation or office hours.
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 an incorrect version before the deadline and realizing that you have done so after the deadline will be counted as a late submission.

Compile and run your code:

After downloading and unzipping the file for assignment3, in your command-line window, `cd` into the directory which contains `src` . You should write your code in `src/eval.ml` .

Use `make` to compile the code. Then, use `./imp` to execute it.

Submission:

Please submit to Canvas with `eval.ml` only.

Feel free to write helper functions in your code.

Feel free to use any library functions in your code.

Operational Semantics

In this problem, you will implement a definitional interpreter (aka an evaluator) for Imp, an imperative programming language. The language contains variables, integer and boolean types, equality and comparison operations, math and boolean operations, and simple control flow. You will also add lambda functions to this imperative programming language. A lambda function can be higher-order. It may take another function as input or return a new function. Lambda functions have been added to C++ (since C++11) and Java (since Java8). It would be nice to support it in Imp!

Recall from class that a definitional interpreter is a function that takes a program's abstract syntax tree (AST) and evaluates it to its final result. For Imp, this final result is an environment, which is a map from variables to their (current) values. Imp's AST is defined by the type `com` (commands), and environments are defined by the type `environment` in the file `ast.ml`. Your job will be to implement the function

```
eval_command : com -> environment -> environment
```

which will evaluate the given command (the first parameter) starting with the given environment (the second parameter) and produce the final environment. Since commands operate over (arithmetic, lambda and other) expressions, you will also implement the function

```
eval_expr : exp -> environment -> value
```

This function evaluates the given expression (the first parameter) to its final value under the given environment (the second parameter). Both of your functions will go in the file `eval.ml`; you should not modify any other files that are given to you (you can add your own testcases).

ast.ml: Abstract Syntax Trees, Environments, Values

As mentioned above, the Imp AST is defined by the type `com` in `ast.ml`. This type has the following definition:

```
type com =
  | While of exp * com
  | For   of exp * com
  | Cond of exp * com * com
  | Comp of com * com
  | Assg of string * exp
  | Declare of dtype * string
  | Skip
```

Each part of this datatype corresponds to a kind of Imp command. For example, `Skip` is an empty command. `Assg of string * exp` corresponds to an assignment command, where the `string` part indicates a variable name, and the `exp` part indicates the expression whose value should be assigned to the variable. `Comp of com * com` is a command that is itself a pair of other commands, with the first executed before the second. We explain this AST in detail in the next section. Note that the types `exp` and `dtype` are part of the Imp AST too (defined in `ast.ml`); they are referenced in `com`'s definition above.

```
type dtype =
  | Int_Type
  | Bool_Type
  | Lambda_Type
```

The interpreter supports integer type `int` as `Int_Type`, bool type `bool` as `Boo_Type`, and lambda function type `lambda` as `Lambda_Type`. Note that we simplify the type system as the type of any lambda function is always `lambda` in the Imp language (examples below).

In a full-fledged interpreter, a parser is used to convert a normal text file into its corresponding AST. In this project, we provide a parser for you. For example, consider the following input file:

```
int n;
int f;
int k;
lambda g;
n := 5;
f := 1;
k := 0;
g := fun x -> fun y -> x + y;
while (n > 1) {
  f := f * n;
  n := n - 1;
  k := g f k;
};
```

The parser will take this and produce the following `com` :

```
Comp(Declare (Int_Type, "n"),
    Comp(Declare (Int_Type, "f"),
        Comp (Declare (Int_Type, "k"),
            Comp (Declare (Lambda_Type, "g"),
                Comp (Assg ("n", Number 5),
                Comp (Assg ("f", Number 1),
                Comp (Assg ("k", Number 0),
                Comp (Assg ("g", Fun ("x", Fun ("y", Plus (Var "x", Var "y")))),
                    While (Lt (Number 1, Var "n"),
                        Comp(Assg ("f", Times (Var "f", Var "n")),
                        Comp(Assg ("n", Minus (Var "n", Number 1)),
                            Assg ("k", App (App (Var "g", Var "f"), Var "k")))))))))))))
```

This `com` uses `Comp` to string together each of the commands in the file, one for each line. The `Declare(Int_Type, "n")` part corresponds to the first line `int n` . The `Declare (Lambda_Type, "g")` part corresponds to the fourth line `lambda g` - the variable `g` refers / will refer to a lambda function. The `Assg("n", Number 5)` corresponds to the third line `n := 5` . And so on.

We suggest before coding you look at the Imp input examples in `/programs` . You should be able to get a sense of how these examples correspond to the `com` type above.

Also in `ast.ml` are the definitions of type value and environment. The former is the result of evaluating an expression (i.e., something of type `exp`). The latter is a map from variables to values; it keeps track of the current value assigned to a given variable. Here are their definitions:

```
type value =
  | Int_Val of int
  | Bool_Val of bool
  | Closure of environment * string * exp
type environment = (string * value) list
```

A value (the result of evaluating an expression) is either an integer, a boolean or a function. The more interesting case is for closures that correspond to function values. `Closure (env, "x", e)` (see Lectures 2/22 and 3/3) represents a function with argument `x` and body-expression `e` that was defined in an environment `env` . We support functions with more than one argument via currying.

An environment is a list of pairs (like a phone-book), where the first element is a variable name and the second is its current value. This representation is called an association list - the first element of the pair is associated with the second. Elements earlier in the list override elements later in the list (we use static scope for Imp). The `List` module has many useful functions for working with association lists which you should consider using in your implementation.

eval.ml: The Evaluator

To implement the definitional interpreter (i.e., the evaluator) you must implement two functions, `eval_expr` and `eval_com` in the file `eval.ml` . This is the only file you should modify.

```
eval_com : com -> environment -> environment
```

```
eval_expr : exp -> environment -> value
```

Each of these takes as an argument an environment. Evaluating a command might modify an environment (e.g., due to a variable assignment), so `eval_com` produces an environment as output. Evaluating an expression will never change the environment, and so `eval_expr` only produces the final value. Both take an environment as input in which the evaluator can look up current values of variables.

To see these functions in action, to give you a sense of what they do, consider some elements from our example Imp program above.

First, consider the first command `Declare (Int_Type, "n")` of type `com` . If we evaluate it in an empty environment `[]` , we should get an output environment `[("n", Int_Val 0)]` . That is,

```
eval_com [] (Declare (Int_Type, "n")) = [("n", Int_Val 0)]
```

An integer variable `n` is initialized to 0 by the interpreter.

Now, consider `Assgn("n", Number 5)` , which is also an Imp command `com` . We evaluate it in an environment `[("f", Int_Val 0); ("n", Int_Val 0)]` . We will get an output environment where we have `n` mapped to `Int_Val 5` . That is,

```
eval_com (Assign("n", Number 5)) [("f", Int_Val 0); ("n", Int_Val 0)] = [("n",Int_Val 5); ("f", I
nt_Val 0); ("n", Int_Val 0)]
```

Now consider `Lt (Number 1, Var "n")` , which is an Imp expression of type `exp` . Evaluating it in the environment produced above, we would have

```
eval_expr (Lt (Number 1, Var "n")) [("n", Int_Val 5); ("f", Int_Val 0); ("n", Int_Val 0)] = Bool_
Val true
```

i.e., `n = 5` is indeed greater than 1.

In what follows, we will step through each of the variants of the `exp` and `com` types to say what your interpreter should do with them.

The interpreter concerns error cases as well such as addition between a boolean and an integer and therefore represent a stuck reduction. The expected behavior for these irreducible error cases can be boiled down to the following rules:

- Any expression containing division by zero should raise a `DivByZero` error when evaluated.
- Any expression or command that is applied to the wrong types should raise a `TypeError` exception when evaluated, for example, the expression 1 + true would result in `TypeError` .
- An expression or command that assigns to an undefined variable, or reads from an undefined variable should raise a `UndefinedVar` when evaluated.
- The above exceptions are defined in `eval.ml` . Evaluation of subexpressions should be done from left to right in order to ensure that lines with multiple possible errors match up with our expected errors.

Function 1: eval_expr

`eval_expr` takes an expression `e` and an environment `env`, and it produces the result of evaluating `e`, which is something of type value (`Int_Val` or `Bool_Val`). Here's what to do with each element of the `exp` type:

Number

Integer literals evaluate to an `Int_Val` of the same value.

True, False

Boolean literals evaluate to a `Bool_Val` of the same value.

Var

An identifier (variable) evaluates to whatever value it is mapped to by the environment. Should raise a `UndefinedVar` if the identifier has no binding.

Plus, Minus, Times, Div, and Mod

These mathematical operations operate only on integers and produce an `Int_Val` containing the result of the operation. All operators must work for all possible integer values, positive or negative, except for division, which will raise a `DivByZeroError` exception in an attempt to divide by zero. If either argument to one of these operators evaluates to a non-integer, a `TypeError` should be raised.

Or and And

These logical operations operate only on booleans and produce a `Bool_Val` containing the result of the operation. If either argument to one of these operators evaluates to a non-boolean, a `TypeError` should be raised.

Not

The unary not operator operates only on booleans and produces a `Bool_Val` containing the negated value of the contained expression. If the expression in the `Not` is not a boolean (and does not evaluate to a boolean), a `TypeError` should be raised.

Less (Lt), LessEqual (Leq)

These relational operators operate only on integers and produce a `Bool_Val` containing the result of the operation. If either argument to one of these operators evaluates to a non-integer, a `TypeError` should be raised.

Equal (Eq)

The equality operator operates both on integers and booleans, but both subexpressions must be of the same type. The operators produce a `Bool_Val` containing the result of the operation. If the two arguments to the operator do not evaluate to the same type (i.e. one boolean and one integer), a `TypeError` should be raised.

Lambda Function (Fun) and Function Application (App)

```
type expr = ...
        | App of exp * exp
        | Fun of string * exp
```

Naturally, to handle functions, the set of values yielded by your evaluator must include closures.

```
type value = ...
          | Closure of environment * string * exp
```

Recall that `App(e1, e2)` corresponds to the application expression `e1 e2` (i.e. applying the argument `e2` to the function `e1`), and `Fun ("x", e)` corresponds to the function defined as `fun x -> e` .

We assume the functions are not recursive. However, a function expression `Fun ("x", e)` does have a value represented by `Closure (env, x, e)` where `env` is the environment at the point that function was declared, and `x` and `e` are the formal parameter and body expression of the function.

For example, you should get the following behavior:

```
(*
    eval_com
        (int y;
        lambda f;
        y := 5;
        f := fun x -> x + y;
        y := 0;
        ) []
    =
        [("y", Number 0),
        ("f", Closure ([("y", 5), ...], "x", Plus (Var "x", Var "y"))),
        ("y", Number 5),
        ...]
*)
let p0 = Declare (Int_Type, "y") in
let p1 = Declare (Lambda_Type, "f") in
let p2 = Assg ("y", Number 5) in
let p3 = Assg ("f", Fun ("x", Plus (Var "x", Var "y"))) in
let p4 = Assg ("y", Number 0) in
let p = Comp (p0, Comp (p1, Comp (p2, Comp (p3, p4)))) in
let env = eval_com p [] in (* env(f) = Closure ([("y", 5), ...], "x", Plus (Var "x", Var "y")) *)
...
```

In the above example, in the environment `env` , the value of `f` is `Closure ([("y", 5), ...], "x", Plus (Var "x", Var "y"))` . The closure has the environment under which `f` was declared; specifically, `y = 5` in that environment despite the reassignment of `y` to 0 after the declaration of `f` . The closure records the formal parameter of `f` , which is `x` . The closure also records the body expression of `f` . The reason that a closure records the environment at the point a function is declared is capturing the static scope of the function. Now suppose we execute `f 3` . As `env(f) = Closure ([("y", 5), ...], "x", Plus (Var "x", Var "y"))` , we have `x = 3` as the actual argument and `y = 5` checked out from the environment in the closure. Thus, `f 3` should return `8` . In other words, `f 3` must find the value of `y` from the static scope of `f` .

For a function application `App(e1, e2)` , your interpreter evaluates `e1` to a value or, more specifically, a closure `Closure (environment, x, e)` , evaluates `e2` to a value `v` , and finally evaluates the body expression `e` under a suitable environment `evn` based on the closure where `evn[x] = v` . See the above example regarding how we execute `f 3` . You should get the following behavior:

```
(* eval
    (fun z -> z - 2) ((fun x -> x + 1) 2)
   =
    Number 1
*)
eval_expr (App (Fun ("z", Minus (Var "z", Number 2)), App (Fun ("x", Plus (Var "x", Number 1)), N
umber 2))) [] = Number 1
```

```
(*
    eval_com (
        int result;
        lambda f;
        lambda g;
        f := fun h -> fun x -> h (h x);
        g := fun x -> x + 3;
        result := f g 5;) []
    =
        [("result", Number 11),
        ("g", Closure ([...], "x", Plus (Var "x", Number 3))),
        ("f", Closure ([...], "h", Fun ("x", App (Var "h", App (Var "h", Var "x"))))),
        ...]
*)
let p0 = Declare (Int_Type, "result") in
let p1 = Declare (Lambda_Type, "f") in
let p2 = Declare (Lambda_Type, "g") in
let p3 = Assg ("f", Fun ("h", Fun ("x", App (Var "h", App (Var "h", Var "x"))))) in
let p4 = Assg ("g", Fun ("x", Plus (Var "x", Number 3))) in
let p5 = Assg ("result", App (App (Var "f", Var "g"), Number 5)) in
let p =  Comp (p0, Comp (p1, Comp (p2, Comp (p3, Comp (p4, p5))))) in
let env = eval_com p [] in (* env[result] = 11 *)
...
```

In the above example, the interpreter uses closures to support the interpretation of the higher-order function `f`. After the evaluation, the value of `result` in the resulting environment is 11.

Function 2: eval_com

`eval_com` takes a command `c` and an environment `env` and produces an updated environment (defined in Types) as a result.

Skip

`Skip` is short for "no operation" and should do just that - nothing at all. The environment should be returned unchanged when evaluating a `Skip`.

Comp

The `Comp` sequencing command is used to compose whole programs as a series of commands. When evaluating `Comp`, evaluate the first subcommand under the environment `env` to create an updated environment `env'`. Then, evaluate the second subcommand under `env'`, returning the resulting environment.

Declare

The `Declare` command is used to create new variables in the environment. If the type being declared is `Int_Type`, a new binding to the value `Int_Val(0)` should be made in the environment. If the type being declared is `Bool_Type`, a new binding to the value `Bool_Val(false)` should be made in the environment. If the type being declared is `Lambda_Type`, a new binding to the value `Closure (env, "x", Var "x")` should be made in the environment where `env` is the underlying environment. This means by default a lambda function is an identity function. The updated environment should be returned.

Assg

The `Assg` command assigns new values to already-declared variables. If the variable hasn't been declared before the assignment, a `UndefinedVar` should be raised. If the variable has been declared to a different type than the one being assigned to it, a `TypeError` should be raised. Otherwise, the environment should be updated to reflect the new value of the given variable, and an updated environment should be returned.

Cond

The `Cond` command consists of three components - a guard expression, an if-body command and an else-body command. The guard expression must evaluate to a boolean - if it does not, a `TypeError` should be raised. If it evaluates to `true`, the if-body should be evaluated. Otherwise, the else-body should be evaluated instead. The environment resulting from evaluating the correct body should be returned.

While

The `While` command consists of two components - a guard expression and a body command. The guard expression must evaluate to a boolean - if it does not, a `TypeError` should be raised. If it evaluates to `true`, the body should be evaluated to produce a new environment and the entire loop should then be evaluated again under this new environment, returning the environment produced by the reevaluation. If the guard evaluates to `false`, the current environment should simply be returned.

For

The `For` loop command consists of two components - a guard expression and a body command. Upon entering the loop, the guard expression a is evaluated in the current environment, yielding an integer `n`. If the int value `n` is 0, the loop body is not executed at all. If `n` is greater than 0, then the loop body is executed `n` times. No action in the body of the loop, such as assigning to a variable, can change the number of times the loop is executed, nor does executing the body alone change the value of any variable except by explicit assignment.

Parser

A real interpreter has two parts: a parser, which takes text files and converts them into an AST, and an evaluator, which runs the AST to produce a final result. Your job has been to implement the evaluator. To help you test this evaluator, we are providing code that will do parsing for you. The function `load` in `assignment3.ml` will take in a text file and produce an AST of type `com`. Our parser (called in `load`) expects an input file whose content is a command (or a sequence of commands, which will be parsed as a single `Comp`). We suggest before coding you look at the Imp input examples in `/programs`. You should be able to get a sense of how these examples correspond to the `com` type above. Our code in the function `eval` in `assignment3.ml` passes the parsed AST to your `eval_command` function with an empty environment.

In the following example, we parse a program from file and apply the `eval` function to obtain the evaluation result of the loaded program. We use a given function `print_env_str` to convert the output environment to string (after removing shadowed variables in the environment). For simplicity, the function `print_env_str` ignores any bindings to closures. The program is:

```
int x;
int y;
x := 5;
x := x + 5;
y := 5 + 5;
y := 5 + x;
```

Obviously, after evaluation, `x = 10` and `y = 15`.

```
In [ ]: let parsed_ast = load ("programs/aexp-add.imp") in
        let result = print_env_str(eval (parsed_ast)) in
        assert(result =
        "- x => 10\n\
         - y => 15\n")
```

One thing you may need to pay attention to for programming the interpreter is nested pattern matching. You will need parentheses to clearly delimit scoping, e.g.,

```
match e with
| Plus (e1, e2) ->
    let r1 = eval_expr e1 env in
    let r2 = eval_expr e2 env in
    (match r1, r2 with
    | Int_Val i, Int_Val j -> ...
    | _ -> ...) (* The parentheses are needed to delimit the scope of the two pattern-matchings *)
| Minus (e1, e2) -> ...
```