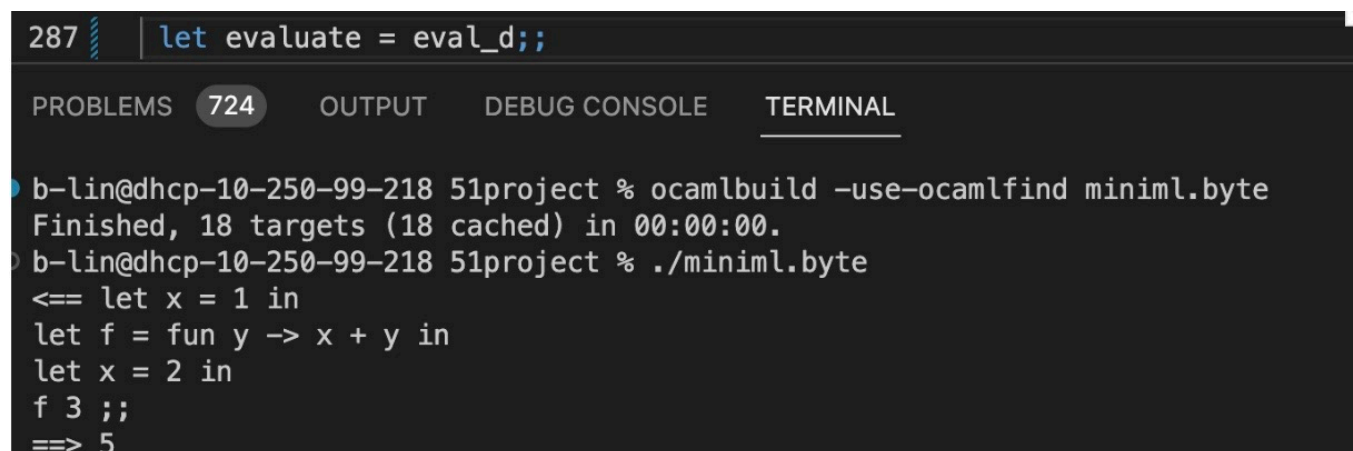


## 1 Introduction

In my final project, I extended MiniML to include lexically scoped environment semantics ( $eval_l$ ), as well as additional operators such as Divide ( $/$ ), GreaterThan ( $>$ ), And ( $\&\&$ ), and Or ( $\|\$ ).

## 2 Lexically Scoped Environment - $eval_l$

Referencing the dynamically scoped environment ( $eval_d$ ), I implemented the Lexically scoped environment to return closures containing the function and the environment when a function was evaluated. Furthermore, I only needed to change the functions for Function (Fun), Letrec, and Functional Application (App) expressions to account for Closures for my lexical environment. Because everything else in my  $eval_d$  and  $eval_l$  are the same, I made a general helper function, called  $eval$ , that takes in the existing function and environment inputs, then added an additional bool input (I called this “lexical”) and used it in the unique expressions (Fun, Letrec, and App) to determine whether it was lexical. If it was, then, it would evaluate the function expressions that were unique to  $eval_l$  and if not, then it would evaluate the function expressions that were unique to  $eval_d$ . So, everytime I called the helper function on itself, I would have to add the additional input “lexical” since  $eval$  takes in three inputs. Here, I followed the Edict of Redundancy, which is to “Never write the same code twice” and it is significant to improving the readability, design, style, scalability, re-usability, and maintainability of my code. To ensure accuracy and correctness of the lexical and dynamic environments, I tested the functions in my parser.



```
287 | let evaluate = eval_d;;  
  
PROBLEMS 724 OUTPUT DEBUG CONSOLE TERMINAL  
  
b-lin@dhcp-10-250-99-218 51project % ocamlbuild -use-ocamlfind miniml.byte  
Finished, 18 targets (18 cached) in 00:00:00.  
b-lin@dhcp-10-250-99-218 51project % ./miniml.byte  
<== let x = 1 in  
let f = fun y -> x + y in  
let x = 2 in  
f 3 ;;  
==> 5
```

Figure 1: Figure 1: An expression evaluated using the Dynamically Scoped Environment. This DOES NOT accord with OCaml’s evaluation.

In  $eval_l$ , the Fun, Letrec, and App cases returns a closure  $Env.Closure$  that contains both the function and the environment at the time of function creation. This closure captures the lexical environment and allows the function body to be evaluated in the captured environment when the function is applied later.

```
287 | let evaluate = eval_l;;

PROBLEMS 724 OUTPUT DEBUG CONSOLE TERMINAL

b-lin@dhcp-10-250-99-218 51project % ocamlbuild -use-ocamlfind miniml.byte
Finished, 18 targets (18 cached) in 00:00:00.
b-lin@dhcp-10-250-99-218 51project % ./miniml.byte
<== let x = 1 in
let f = fun y -> x + y in
let x = 2 in
f 3 ;;
==> 4
```

Figure 2: An expression evaluated using the Lexically Scoped Environment. This DOES accord to OCaml's evaluation.

In Figure 2, we see that  $x$  in “let  $x = 1$ ” is created/defined so when we evaluate the function  $f\ 3$ , we get  $1 + 3 = 4$ . On the other hand, in `eval_d`, the cases simply returns the function itself without capturing the environment, i.e., the environment when the function is applied. In Figure 1, we see that the output is 5. This is because the environment at the time of function application is when  $x = 2$ . Therefore, when we call  $f\ 3$  we get  $2 + 3 = 5$ . By incorporating these differences, `eval_l` enables lexical scoping and ensures that the function body is evaluated in the correct lexical environment.

### 3 Additional Atomic Types and Operators

The given implementation of `binop` in the `expr.ml` file is missing a few operators so I added the Divide (`/`), GreaterThan (`>`), And (`&&`), and Or (`||`) operators. I added these operators into my `expr.ml` file within the `binop` algebraic data type, the `Binop` case within my `exp_to_concrete_string` function, and the `binop_to_string` inner function within my `exp_to_abstract_string` function. Additionally, I also added it to my parser (`miniml_parse.mly` and `miniml_lex.mll`) files. For the `miniml_parse.mly` file, I added the operators as a part of the same categories as `%token` and `%left`. For instance, I added `GREATERTHAN` along with `LESSTHAN` and `EQUALS` and grouped `AND` and `OR` into its own `%token` and `%left` category. Additionally, I the operators as a part of `expnoapp` by using the same structure as the other expressions, and replacing the first input of `Binop` with the added Operator. On top of that, I added the corresponding symbols for each operator within the `sym_table` function:

- (1) `"/` for Divide
- (2) `>` for GreaterThan
- (3) `&&` for And
- (4) `||` for Or

Lastly, I added the other operators in my `evaluation.ml` file to the helper functions (`binopeval` and `unopeval`) implemented in from Lab 9 — shown in Figure 3.

```
| Divide, Num x1, Num x2 -> Num (x1 / x2)
| Divide, _, _ -> raise (IllFormed "can't divide non-integers")

| GreaterThan, Num x1, Num x2 -> Bool (x1 > x2)
| GreaterThan, _, _ -> raise (IllFormed "non-integers not greater than")
| And, Bool x1, Bool x2 -> Bool (x1 && x2)
| And, _, _ -> raise (IllFormed "non-integers not conjoined by and")
| Or, Bool x1, Bool x2 -> Bool (x1 || x2)
| Or, _, _ -> raise (IllFormed "non-integers not conjoined by or");;
```

Figure 3: Additional operator cases added to binopeval. The And and Or cases uses the Bool expression to account for the binary inputs, x1 and x2.

To ensure the accuracy and correctness of these newly added operators, I tested various expressions in my parser using the lexical environment.

```
● b-lin@dhcp-10-250-99-218 51project % ocamlbuild -use-ocamlfind miniml.byte
  Finished, 18 targets (18 cached) in 00:00:00.
○ b-lin@dhcp-10-250-99-218 51project % ./miniml.byte
<== 50 / 10;;
==> 5
<== 50 > 10;;
==> true
<== 50 < 10;;
==> false
<== (true && true);;
==> true
<== (true && false);;
==> false
<== (50 > 10) && (50 < 10);;
==> false
<== (false || false);;
==> false
<== (true || false);;
==> true
<== (50 > 10) || (50 < 10);;
==> true
```

Figure 4: Expressions conjoined with Divide, GreaterThan, And, and Or operators.