Mateo Velarde
Stuart Shieber
Computer Science 51
May 5th, 2021

Final Project: Implementing MiniML

IMPLEMENTATION OF EXPR.ML

For expr.ml, I implemented free_vars, new_varname, subst, exp_to_concrete_string, and exp_to_abstract_string. Originally, for free_vars I had the returned expression of Let and Letrec to be identical. However, after attending office hours, I realized that Letrec (var, expr1, expr2) also should remove instances of the var in expr1, as well as expr2. After I resolved those minor issues, my future implementation of eval_s worked properly. For the majority of free_vars, I followed the rules outlined in page 251 of the textbook. Following those rules, I had free_vars correctly match the expression that was passed in with all the types of expr, with the last two – Raise and Unassigned – being called using a catch all case to be less redundant with the code. Hence, each match case returned their respective set of free variables using the SS module that was given in the outline of the code.

```
let rec free_vars (exp : expr) : varidset =
  match exp with
  | Var (v) -> SS.singleton v
  | Num (_i) -> SS.empty
  | Unop (_u, ex) -> free_vars ex
  | Binop (_bin, ex1, ex2) -> SS.union (free_vars ex1) (free_vars ex2)
  | Fun (var, ex) -> SS.remove var (free_vars ex)
  | Let (var, expr1, expr2) -> SS.union (free_vars expr1) (SS.remove var (free_vars expr2))
  | Letrec (var, expr1, expr2) -> SS.union (SS.remove var (free_vars expr1)) (SS.remove var (free_vars expr2))
  | App (expr1, expr2) -> SS.union (free_vars expr1) (free_vars expr2)
  | _ -> SS.empty ;;
```

For new_varname, I copied the function gensym from lab 10. The use of gensym is to receive a string and use a reference of int set to 0 to continuously return the string attached to the int. Each time the function is called, it would increment the reference of int, thus returning a new string each time new_varname is called. The use of gensym from lab aided quite a lot in the implementation.

```
let new_varname () : varid =
  let gensym : string -> string =
    fun s ->
      let origin = !space in
      space := !space + 1;
      s ^ string_of_int origin
  in gensym "var" ;;
```

What follows next is subst, which required quite a lot of testing and manipulation to obtain a working implementation. For mostly match case, I followed the rules outlined in page 252 from the textbook. First, I changed the function to be recursive. After implementing the rules from the textbook all that required debugging was the implementation of the Letrec match case. Originally, I had the match case of Let and Letrec be the same. However, the error came when running eval_s later in my progress of completing the final project. Through office hours and debugging by using print statements, I was able to change the implementation of Letrec to return a new Letrec with its definition set to the substitution of var_name, repl, and another substitution of var, a new variable – using the new_varname function – and the definition. The body of the Letrec that is returned is basically the same line of code as the definition, except that rather than the definition being the substituted body, it is the original body of the original Letrec. Other than that obstacle in the implementation of subst, the rest of the match cases more or less called subst again with their expression. Similarly to free_vars, the match cases for Raise and Unassigned were captured in a capture-all match case that just returned the same expression.

```
let rec subst (var_name : varid) (repl : expr) (exp : expr) : expr =
  match exp with
  | Var x -> if x = var_name then repl else exp
  | Num _n -> exp
  | Unop(op, arg) -> Unop(op, subst var_name repl arg)
  | Binop(op, arg1, arg2) -> Binop(op, subst var_name repl arg1, subst var_name repl arg2)
  | Let(y, def, body) ->
    if y <> var_name && not(SS.mem y (free_vars repl)) then
      Let(y, subst var_name repl def, subst var_name repl body)
    else if y <> var_name && SS.mem y (free_vars repl) then
      let newvar = new_varname () in
      Let(newvar, subst var_name repl (subst y (Var newvar) def),
                  subst var_name repl (subst y (Var newvar) body))
    else
      Let(y, subst var_name repl def, body)
  | Fun (var, ex) ->
    if var <> var_name && not(SS.mem var (free_vars repl)) then
      Fun(var, subst var_name repl ex)
    else if var <> var_name && SS.mem var (free_vars repl) then
      let newvar = new_varname () in
      Fun(newvar, subst var_name repl (subst var (Var newvar) ex))
    else
      exp
  | Letrec (x, def, body) ->
    if x <> var_name && not(SS.mem x (free_vars repl)) then
      Letrec(x, subst var_name repl def, subst var_name repl body)
    else if x <> var_name && SS.mem x (free_vars repl) then
      let newvar = new_varname () in
      Letrec(newvar, subst var_name repl (subst x (Var newvar) def),
                     subst var_name repl (subst x (Var newvar) body))
    else
      exp
  | App (expr1, expr2) -> App(subst var_name repl expr1, subst var_name repl expr2)
  | Conditional (exif, exyes, exno) -> Conditional (subst var_name repl exif,
                                                    subst var_name repl exyes,
                                                    subst var_name repl exno)
  | _ -> exp
;;
```

The final two functions implemented in expr.ml were exp_to_concrete_string and exp_to_abstract_string. Firstly, I converted both functions into recursive functions. Next, following the descriptions of concrete and abstract syntax from the beginning of the textbook, I matched the expression passed into both functions to all types of expr. For exp_to_concrete_string, I extracted the value inside each expression and retuned it as a string using built in string functions, such as string_of_int. In the cases of Raise and Unassigned, I returned a string of both types. With exp_to_abstract_string, I expanded upon the implementations I had for exp_to_concrete_string to account for the string representations of the expressions, such as “Var” and “Num.” Each expression uses parentheses to symbolize what lies in the scope of the expression. For match cases that hold more expressions, I called the functions again on the extracted expression, thus illustrating the purpose of the recursive implementations.

```ocaml
let rec exp_to_concrete_string (exp : expr) : string =
  match exp with
  | Var (v) -> v
  | Num (i) -> string_of_int i
  | Bool (b) -> string_of_bool b
  | Unop (u, ex) -> (match u with
                     | Negate -> " ~- " ^ exp_to_concrete_string ex)
  | Binop (bin, ex1, ex2) ->
    (match bin with
    | Plus -> exp_to_concrete_string ex1 ^ " + " ^ exp_to_concrete_string ex2
    | Minus -> exp_to_concrete_string ex1 ^ " - " ^ exp_to_concrete_string ex2
    | Times -> exp_to_concrete_string ex1 ^ " * " ^ exp_to_concrete_string ex2
    | Equals -> exp_to_concrete_string ex1 ^ " = " ^ exp_to_concrete_string ex2
    | LessThan -> exp_to_concrete_string ex1 ^ " < " ^ exp_to_concrete_string ex2 )
  | Conditional (exif, exyes, exno) ->
    "if " ^ exp_to_concrete_string exif ^ " then " ^ exp_to_concrete_string exyes ^
    " else " ^ exp_to_concrete_string exno
  | Fun (var, ex) -> var ^ exp_to_concrete_string ex
  | Let (var, expr1, expr2) ->
    "let " ^ var ^ exp_to_concrete_string expr1 ^ " = " ^ exp_to_concrete_string expr2
  | Letrec (var, expr1, expr2) ->
    "let rec " ^ var ^ exp_to_concrete_string expr1 ^ " = " ^ exp_to_concrete_string expr2
  | Raise -> "raise "
  | Unassigned -> "unassigned "
  | App (expr1, expr2) ->
    exp_to_concrete_string expr1 ^ " " ^ exp_to_concrete_string expr2
  ;;
```

```ocaml
let rec exp_to_abstract_string (exp : expr) : string =
  match exp with
  | Var (v) -> "Var(" ^ v ^ ")"
  | Num (i) -> "Num(" ^ string_of_int i ^ ")"
  | Bool (b) -> "Bool(" ^ string_of_bool b ^ ")"
  | Unop (u, ex) -> (match u with
                     | Negate -> "Negate(" ^ exp_to_abstract_string ex ^ ")" )
  | Binop (bin, ex1, ex2) ->
    let sign = match bin with
               | Plus -> "Plus"
               | Minus -> "Minus"
               | Times -> "Times"
               | Equals -> "Equals"
               | LessThan -> "LessThan"
    in "Binop(" ^ sign ^ ", " ^ exp_to_abstract_string ex1 ^ ", " ^
       exp_to_abstract_string ex2 ^ ")"
  | Conditional (exif, exyes, exno) ->
    "Conditional(" ^ exp_to_abstract_string exif ^ ", " ^ exp_to_abstract_string exyes ^
    ", " ^ exp_to_abstract_string exno ^ ")"
  | Fun (var, ex) -> "Fun(" ^ var ^ ", " ^ exp_to_abstract_string ex ^ ")"
  | Let (var, expr1, expr2) ->
    "Let(" ^ var ^ ", " ^ exp_to_abstract_string expr1 ^ ", " ^
    exp_to_abstract_string expr2 ^ ")"
  | Letrec (var, expr1, expr2) ->
    "Letrec(" ^ var ^ ", " ^ exp_to_abstract_string expr1 ^ ", " ^
    exp_to_abstract_string expr2 ^ ")"
  | Raise -> "Raise "
  | Unassigned -> "Unassigned "
  | App (expr1, expr2) ->
    "App(" ^ exp_to_abstract_string expr1 ^ ", " ^ exp_to_abstract_string expr2 ^ ")"
  ;;
```

IMPLEMENTATION OF EVALUATION.ML

For evaluation.ml, I completed programming the functions for the Env module, substitution evaluation, dynamic evaluation, and lexical evaluation. In terms of the close, and lookup functions in the Env module, they were quite easy to implement based the description of environments on chapter 19 of the textbook. A conceptual problem I encountered when implementing lookup and extend pertained to its use. Initially, I had the extend function search

for varname in the environment that is passed in and replace it with the value reference. However, thanks to Professor Shieber's guidance, I was able to comprehend that the function merely adds the tuple of varname and value reference to the environment list. This small error did help implementing dynamic evaluation. Proceeding extend was value_to_string and env_to_string, which I joined together using the "and" keyword for functions that rely on each other. I used match cases on both functions to the respective types of the argument. The "and" keyword came useful in env_to_string when calling value_to_string when returning the string of the extracted value. Consequently, value_to_string calls env_to_string when trying to obtain the string representation of the environment that is passed in the Closure match case for type value.

```
let lookup (env : env) (varname : varid) : value =
  if List.mem_assoc varname env then !(List.assoc varname env)
  else raise (EvalError "Variable not found")

let extend (env : env) (varname : varid) (loc : value ref) : env =
  (varname, loc) :: (List.remove_assoc varname env)

let rec value_to_string ?(printenvp : bool = true) (v : value) : string =
  match v with
  | Val x -> exp_to_concrete_string x
  | Closure (exp, env) ->
    if printenvp = false then exp_to_concrete_string exp
    else exp_to_concrete_string exp ^ env_to_string env
and env_to_string (env : env) : string =
  match env with
  | [] -> ""
  | (varid1, value1):: tl ->
    varid1 ^ ", " ^ value_to_string !value1 ^ " and " ^ env_to_string tl
```

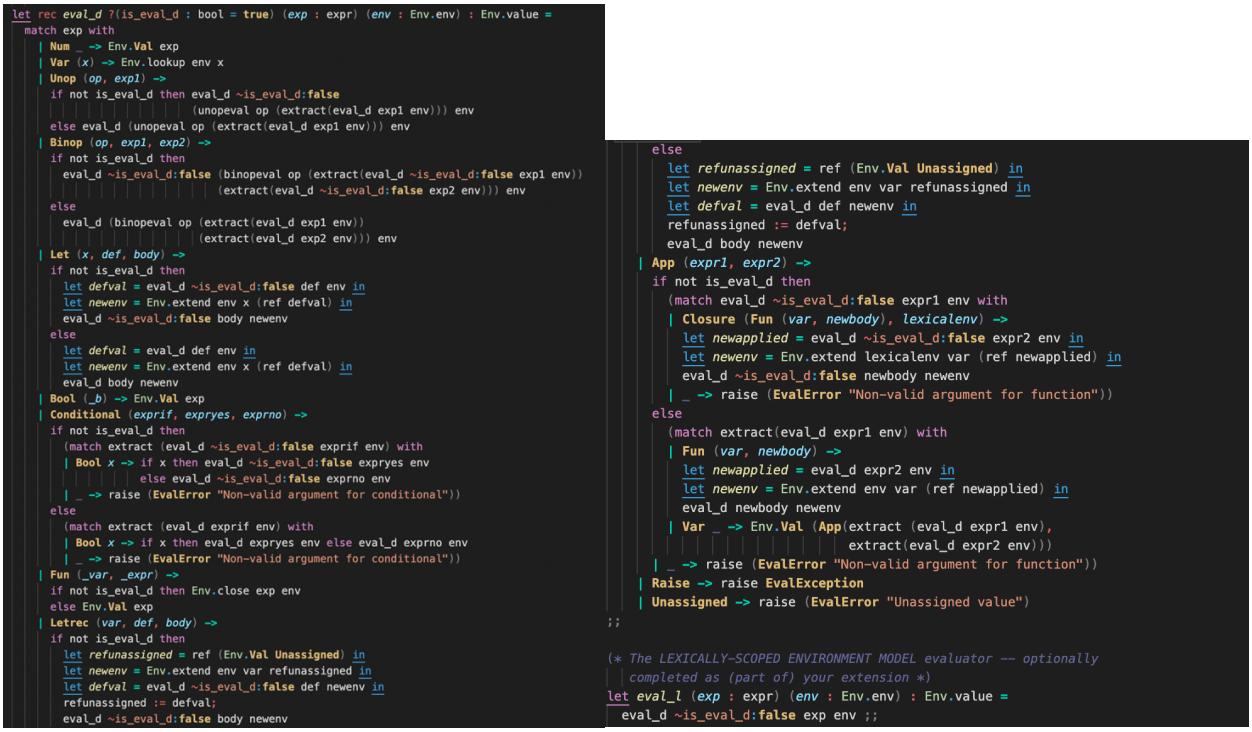
For all evaluations, I created global functions that evaluated expressions of binary operations, unary operations, and extracting expressions from values of the Env module. For substitution evaluation, I followed the substitution evaluation rules from page 255 from the textbook. To make the recursive calls less redundant, I created a recursive function inside eval_s to just take in an expression. In the new recursive function, I matched the expression with all the types of expression. Binop and Unop would make use of the global functions, Conditional would extract the value of the first expression to see if it should evaluate the first expression or the next. Let and App also just followed the rules from the textbook which just made use of the subst function from expr.ml to substitute evaluated definitions to bodies. The second-largest problem when implementing eval_s was trying to find the format Letrec should evaluate. Through office

hours and Professor Shieber's video, I was able to obtain the rule for Letrec, which substitutes a new expression Var into a new expression Letrec into the body of the original evaluated body of the Letrec. Following that change allowed eval_s to properly work. Furthermore, once the recursive helper function inside eval_s retuned the evaluated expression, I used Env.Val to convert the expression into a value such that the return type for eval_s matched.

```
let eval_s (exp : expr) (_env : Env.env) : Env.value =
  let rec eval (exp: expr): expr =
    match exp with
    | Num _ -> exp
    | Var (x) -> raise (EvalError ("Unbound variable " ^ x))
    | Unop (op, exp1) -> unopeval op (eval exp1)
    | Binop (op, exp1, exp2) -> binopeval op (eval exp1) (eval exp2)
    | Let (x, def, body) -> eval (subst x (eval def) body)
    | Bool (_b) -> exp
    | Conditional (exprif, expryes, exprno) ->
      (match eval exprif with
      | Bool x -> if x then eval expryes else eval exprno
      | _ -> raise (EvalError "Non-valid argument for conditional"))
    | Fun (_var, _expr) -> exp
    | Letrec (var, def, body) ->
      eval (subst var (subst var (Letrec (var, (eval def), Var (var)))
                (eval def)) body)
    | App (expr1, expr2) ->
      (match eval expr1 with
      | Fun (var, ex) -> eval (subst var (eval expr2) ex)
      | _ -> raise (EvalError "Non-valid argument for function"))
    | Raise -> raise EvalException
    | Unassigned -> raise (EvalError "Unassigned value")
  in
  Env.Val (eval exp) ;;
```

Finally, for eval_d and eval_l, the extension I implemented, I will discuss both at the same time since they were astonishingly similar. Initially, both eval_d and eval_l had respective match cases to match with the valuation rules as described on pages 403 and 397. I merely followed the evaluation rules for eval_d from the textbook, except for when it came to Letrec, which was not explicitly defined. Through trial and error, as well as confirmation from office hours, I was able to comprehend that the Letrec match case had to create an Unassigned reference to extend an environment, evaluate the definition of Letrec with that environment, and then re-reference the Unassigned reference to that new value. Hence, after those three steps were executed, Letrec evaluated the body with the new environment. When implementing lexical evaluation, I had the same implementations for every match case except for Fun and App. To adhere to the Edict of Redundancy, I added an optional argument to eval_d of type bool –

initially set to true – which would determine whether the function was evaluating dynamically or lexically. In the case of dynamic evaluation, there would be no optional argument evoked and the function would work as described above. When evaluating lexically, eval_l would pass the optional argument set to false, an expression, and an environment. Hence, eval_s was modified in nearly all match cases to account for a false optional argument and change the return value of the expression. The implementation of Fun for lexical evaluation involved invoking the close function of the Env module. Additionally, the implementation of App accounted for having to use the lexical environment when evaluating the original body from the extracted function. Therefore, by using the optional argument I was able to reduce code and find similarities.



```
let rec eval_d ?(is_eval_d : bool = true) (exp : expr) (env : Env.env) : Env.value =
  match exp with
    | Num _ -> Env.Val exp
    | Var (x) -> Env.lookup env x
    | Unop (op, exp1) ->
      if not is_eval_d then eval_d ~is_eval_d:false
                              (unopeval op (extract(eval_d exp1 env))) env
      else eval_d (unopeval op (extract(eval_d exp1 env))) env
    | Binop (op, exp1, exp2) ->
      if not is_eval_d then
        eval_d ~is_eval_d:false (binopeval op (extract(eval_d ~is_eval_d:false exp1 env))
                                 (extract(eval_d ~is_eval_d:false exp2 env))) env
      else
        eval_d (binopeval op (extract(eval_d exp1 env))
                             (extract(eval_d exp2 env))) env
    | Let (x, def, body) ->
      if not is_eval_d then
        let defval = eval_d ~is_eval_d:false def env in
        let newenv = Env.extend env x (ref defval) in
        eval_d ~is_eval_d:false body newenv
      else
        let defval = eval_d def env in
        let newenv = Env.extend env x (ref defval) in
        eval_d body newenv
    | Bool (_b) -> Env.Val exp
    | Conditional (exprif, expryes, exprno) ->
      if not is_eval_d then
        (match extract (eval_d ~is_eval_d:false exprif env) with
        | Bool x -> if x then eval_d ~is_eval_d:false expryes env
                        else eval_d ~is_eval_d:false exprno env
        | _ -> raise (EvalError "Non-valid argument for conditional"))
      else
        (match extract (eval_d exprif env) with
        | Bool x -> if x then eval_d expryes env else eval_d exprno env
        | _ -> raise (EvalError "Non-valid argument for conditional"))
    | Fun (_var, _expr) ->
      if not is_eval_d then Env.close exp env
      else Env.Val exp
    | Letrec (var, def, body) ->
      if not is_eval_d then
        let refunassigned = ref (Env.Val Unassigned) in
        let newenv = Env.extend env var refunassigned in
        let defval = eval_d ~is_eval_d:false def newenv in
        refunassigned := defval;
        eval_d ~is_eval_d:false body newenv
      else
        let refunassigned = ref (Env.Val Unassigned) in
        let newenv = Env.extend env var refunassigned in
        let defval = eval_d def newenv in
        refunassigned := defval;
        eval_d body newenv
    | App (expr1, expr2) ->
      if not is_eval_d then
        (match eval_d ~is_eval_d:false expr1 env with
        | Closure (Fun (var, newbody), lexicalenv) ->
          let newapplied = eval_d ~is_eval_d:false expr2 env in
          let newenv = Env.extend lexicalenv var (ref newapplied) in
          eval_d ~is_eval_d:false newbody newenv
        | _ -> raise (EvalError "Non-valid argument for function"))
      else
        (match extract(eval_d expr1 env) with
        | Fun (var, newbody) ->
          let newapplied = eval_d expr2 env in
          let newenv = Env.extend env var (ref newapplied) in
          eval_d newbody newenv
        | Var _ -> Env.Val (App(extract (eval_d expr1 env),
                                extract(eval_d expr2 env)))
        | _ -> raise (EvalError "Non-valid argument for function"))
    | Raise -> raise EvalException
    | Unassigned -> raise (EvalError "Unassigned value")
;;

(* The LEXICALLY-SCOPED ENVIRONMENT MODEL evaluator -- optionally
   completed as (part of) your extension *)
let eval_l (exp : expr) (env : Env.env) : Env.value =
  eval_d ~is_eval_d:false exp env ;;
```

A few last notes for evalution.ml, include the raises of EvalException for all Raise match cases and raise (EvalError “Unassigned value”) for all Unassigned match cases in eval_s, eval_d, eval_l. The final changes to evaluation.ml come from the instantiation of four new variables which correspond to eval_t, eval_s, eval_d, and eval_l which will be referenced in miniml.ml.

TESTING AND MINIML.ML

In miniml.ml, I replicated the same layout as in Professor Shieber's tutorial to display the results of substitution, dynamic, and lexical evaluation. I implemented this by creating four separate try-with unit cases, which would call on the evaluate definitions from evaluation.ml and display the concrete string of their respective evaluations. In terms of testing, found in unit_tests.ml, I used the helper functions from the modules I implemented and checked them to make sure they were working properly. For example, to test that free_vars was working, I used the same_vars function and vars_of_list function to verify that my functions were behaving normally. When putting this into the interpreter, the results would coincide with the results obtained in lab 9.

```
let tests () =
  unit_test (same_vars (free_vars (Let ("x",
                                   Binop (Plus, Var "x", Var "y"),
                                   Binop (Times, Var "z", Num 3))))
                       (vars_of_list ["x"; "y"; "z"]) = true)
                       "free_vars with three variables";
  unit_test (same_vars (free_vars (Let("x",
                                    Num(3), Let("y", Var("x"), App(App(Var("f"), Var("x")), Var("y"))))))
                       (vars_of_list ["f"]) = true)
                       "free_vars with a function";
  unit_test (same_vars (free_vars (Let("x",
                                    Var("x"), Let("y", Var("x"),
                                    App(App(Var("f"), Var("x")), Var("y"))))))
                       (vars_of_list ["f"; "x"]) = true)
                       "free_vars with function and variables";
```

Testing eval_s, eval_d, and eval_l was much better due to having implemented it. Other than the unit_tests, I could also type the expressions into miniml.byte to see the results of such evaluations and determine whether they coincide with the results in the textbook or from Professor Shieber's tutorial.

```
<== let x = 1 in let f = fun y -> x + y in let x = 2 in f 3;;
--> Let(x, Num(1), Let(f, Fun(y, Binop(Plus, Var(x), Var(y))), Let(x, Num(2), App(Var(f), Num(3)))))
s=> 4
d=> 5
l=> 4
```

The tests I implemented accounted for errors, multiple variables, and the results of the same expression for all three types of evaluation. In one particular case, I dynamically evaluated an expression which returns 1, but returns an error for substitution and lexical evaluation. Thus, in unit_tests.ml, I only tested for the dynamic evaluation and manually checked the evaluation

for eval_s and eval_l by typing it into the interpreter, which gave the correct outputs according to Professor Shieber's tutorial.

```
(* let f = fun x -> if x = 0 then 1 else f (x - 1) in f 5;;
  This case has eval_s and eval_l return exception Evaluation.EvalError("Unbound variable f")
  therefore it is not tested *)
unit_test (eval_d (Let("f", Fun("x", Conditional(Binop(Equals, Var("x"), Num(0)),
                    Num(1), App(Var("f"), Binop(Minus, Var("x"), Num(1))))),
                    App(Var("f"), Num(5)))) (Env.empty ())
                    =  Env.Val (Num 1))
                  "eval_d with val 1, where s = l = ERROR and d=1";
```

```
<== let f = fun x -> if x = 0 then 1 else f (x - 1) in f 5;;
--> Let(f, Fun(x, Conditional(Binop(Equals, Var(x), Num(0)), Num(1), App(Var(f), Binop(Minus, Var(x), Num(1))))), App(Var(f), Num(5)))
xx> evaluation error: Unbound variable f
d=> 1
xx> evaluation error: Variable not found
```

CONCLUSION

The implementation of a subset of OCaml did perform as expected in different situations. The substitution semantics were needed for substitution evaluation, while dynamic and lexical evaluations relied on environments to evaluate expressions. While dynamic evaluation had a few advantages, it was not as well-structured in terms of application as lexical evaluation is. All of the unit tests, which determined if the functions of my final project were implemented correctly, passed and changing some of the supplied files, such as miniml.ml came to be quite useful in differentiating results from the three types of evaluation. In terms of extensions, I only implemented lexical evaluation and used optional arguments to reduce redundancy while illustrating the similarities between dynamic and lexical. More extensions could have been applied, such as more representations of types and error-checking, regardless, the miniML my final project presents passes all the requirements as stated in the textbook and lectures. Truly, I had a great time programming in OCaml and learning to abstract different types of code and functions to paradigms I never thought existed, hence showing the beauty and use of functional programming.