CS51 Final Project : MiniML Writeup

Harvard University, Professors Brian Yu and Stephen Chong

Anthony Malysz

May 4, 2022

Introduction

For my required extensions to the base-level subset of the OCaml language that MiniML already supported to this point, I chose to implement the following: 1.) a function that supports evaluations for lexically-scoped environments in addition to the completed dynamically-scoped environment model, 2.) support for floating-point numbers and their relative operators (e.g. +.), 3.) several new unary operators, and 4.) several new binary operators. The goal for my implementations was to build a MiniML that could handle any and all arithmetic expressions, which may or may not be nested within other (let, let rec, fun, …, etc.) expressions.

1.1 Lexically-Scoped Environment Model

What makes a lexically-scoped environment model attractive is its introduction of "closures", so that when functions are called they use the environment in which they were first defined. This is opposed to the dynamically-scoped environment model, which as the name suggests, offers a dynamic application environment such that functions use the most current environment when called.

Both methods offer certain advantages and disadvantages, particularly when it comes to keeping track of which environments certain functions were called and defined within. This is of particular interest in instances where pointers are implemented; unfortunately I did not have enough time to implement them without sacrificing the quality of the remainder of my MiniML, but they would have been my next implementation otherwise. Below you will find a description of eval_s, eval_d, and eval_l.

1.2 eval_l vs. eval_d & eval_s

After writing eval_s and referencing the description for *Stage 8* in the textbook, I came to the realization that not only would eval_d have significant overlap with eval_l (to the point of redundancy), but eval_s, too, would have significant overlap with both of the other eval functions. My solution was to write a parent function, "rec evaluator (sub) (dyna) (lexi) (exp) (env)", that takes in booleans to signify whether it is to evaluate an expression exp in a dyna(mic) or lexi(cal) environment env, or to simply evaluate a (sub)stitution. The functions eval_s, eval_d, and eval_l themselves are simply calls to rec evaluator with (true false false exp env), (false true false exp env), or (false false true exp env) arguments, respectively. This would essentially include (the unfinished) eval_e below.

1.3 Implementation of evaluator

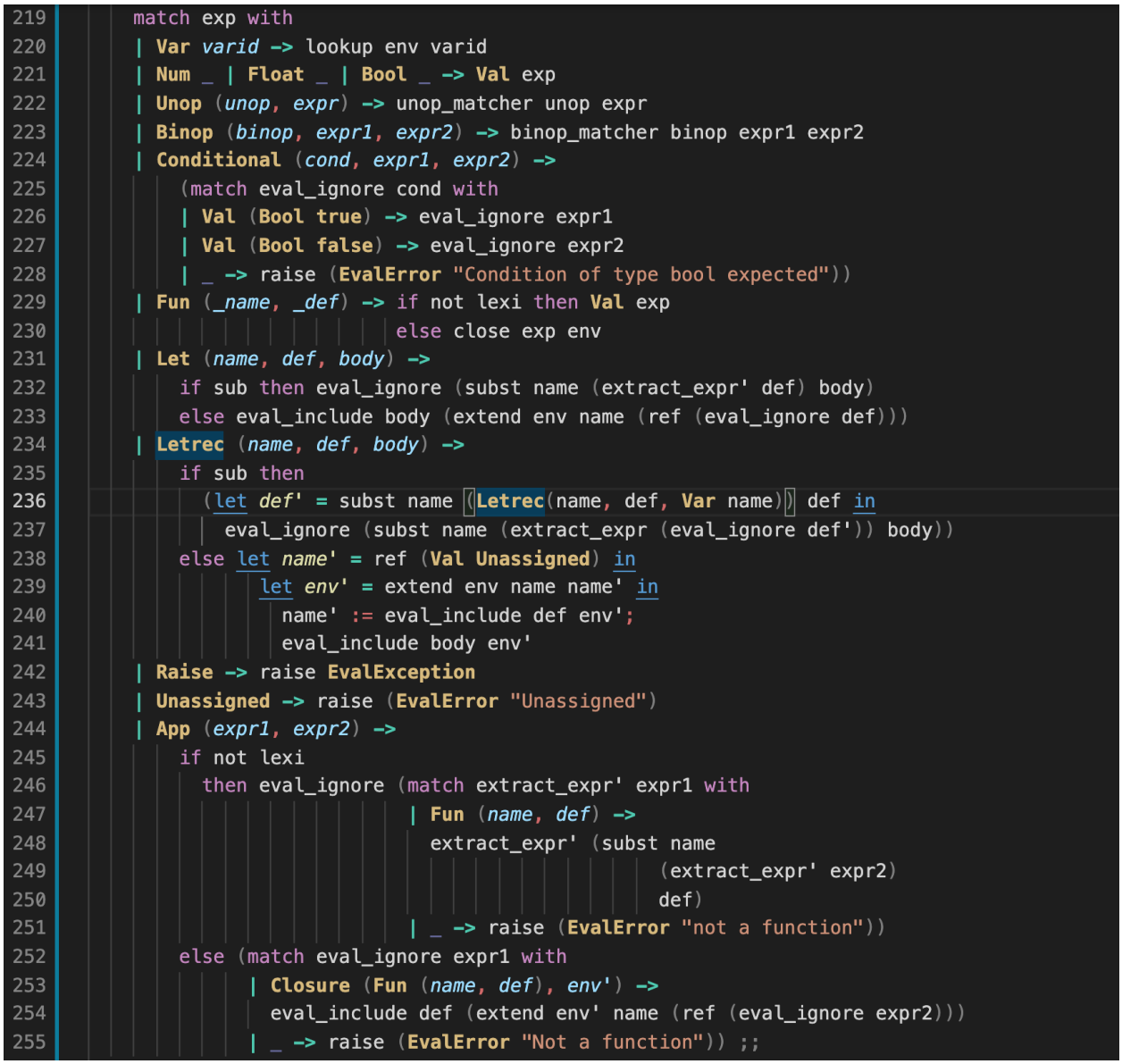
Below is the type signature for evaluator, as stated earlier. Notice that there are several helper functions that serve to either further abstract the components of evaluator (e.g. unop_matcher & binop_matcher) or to simplify and condense its code via shorthand expressions (e.g. eval_ignore & eval_include).

```
145  (* overarching evaluator for every subsequent eval_ *)
146  let rec evaluator (sub : bool)
147                    (dyna: bool)
148                    (lexi: bool) (exp : expr) (env : env) : value =
149
150    (* shorthand for rec call that ignores environment *)
151    let eval_ignore (exp': expr) : value =
152      evaluator sub dyna lexi exp' env in
153
154    (* shorthand for rec call that includes environment *)
155    let eval_include (exp': expr) (env': env) : value =
156      evaluator sub dyna lexi exp' env' in
157
158    (* first seen in expr.ml; see note there about pattern-match redundancy *)
159    let unop_matcher (unop: unop) (expr : expr) : value =
160      match unop, expr with
161      | Negate_i, Num n -> Val(Num(~-n))
162      | Negate_f, Float n -> Val(Float(~-.n))
163      | Not, Bool b -> Val(Bool(not b))
164      | _ -> raise (EvalError "(unop, expr type) combination not supported")
165      (* new unops and expressions go here *) in
166
```

Figure 1: *evaluator type signature, shorthand helpers, & unop_matcher*

Aside from making the actual body of evaluator look much nicer and simpler, writing unop_matcher and binop_matcher serve an additional purpose of allowing for growth. In fact, before I decided to even implement floats and additional operations, I wrote evaluator in this way with the foresight that I would still need to implement additions to the subset of the OCaml language that MiniML would support. This turned out to be incredibly beneficial.

Aside from a few other helper functions not pictured (including binop_matcher, which behaves similarly to unop_matcher, only for binary operators), below is the body of evaluator. It works by matching the inputted expression exp against every possible type of expr, from Var(varid) to App(expr1, expr2). A variable will have its name looked up in its native environment to evaluate it. Any number, including ints and floats, and any bool will simply evaluate to itself since they are universally equivalent. Unops and Binops will call unop_match and binop_match; similarly, Conditional has its own match, but since it will not ever be extended (since a condition can only be true or false), its match remains nested within the body of evaluator. Raise and Unassigned are likewise universally equivalent.



```
219      match exp with
220      | Var varid -> lookup env varid
221      | Num _ | Float _ | Bool _ -> Val exp
222      | Unop (unop, expr) -> unop_matcher unop expr
223      | Binop (binop, expr1, expr2) -> binop_matcher binop expr1 expr2
224      | Conditional (cond, expr1, expr2) ->
225          (match eval_ignore cond with
226          | Val (Bool true) -> eval_ignore expr1
227          | Val (Bool false) -> eval_ignore expr2
228          | _ -> raise (EvalError "Condition of type bool expected"))
229      | Fun (_name, _def) -> if not lexi then Val exp
230                             else close exp env
231      | Let (name, def, body) ->
232          if sub then eval_ignore (subst name (extract_expr' def) body)
233          else eval_include body (extend env name (ref (eval_ignore def)))
234      | Letrec (name, def, body) ->
235          if sub then
236            (let def' = subst name (Letrec(name, def, Var name)) def in
237             eval_ignore (subst name (extract_expr (eval_ignore def')) body))
238          else let name' = ref (Val Unassigned) in
239               let env' = extend env name name' in
240                 name' := eval_include def env';
241                 eval_include body env'
242      | Raise -> raise EvalException
243      | Unassigned -> raise (EvalError "Unassigned")
244      | App (expr1, expr2) ->
245          if not lexi
246            then eval_ignore (match extract_expr' expr1 with
247                              | Fun (name, def) ->
248                                extract_expr' (subst name
249                                                     (extract_expr' expr2)
250                                                     def)
251                              | _ -> raise (EvalError "not a function"))
252          else (match eval_ignore expr1 with
253               | Closure (Fun (name, def), env') ->
254                 eval_include def (extend env' name (ref (eval_ignore expr2)))
255               | _ -> raise (EvalError "Not a function")) ;;
```

Figure 2: *body of rec evaluator function*

Finally, we get to the interesting bits of abstraction. The way evaluator handles different cases for environment scope or substitution is via if/else statements within Fun, Let, Letrec, and App, which are generally the points of interest in OCaml as a whole; this is the functionality which allows us to write such complex–sometimes even infinitely nested–expressions. Within the Fun match case, if the method is not lexical (substitution or dynamic), it will simply return itself; otherwise, it will return a Closure * for exp and its native environment env–in both cases the actual name and definition of the matched Fun in question are irrelevant.

**Note: closures are important because in lexical environments, they create separation between other environments so that functions defined between closures are not able to be defined within another env. This is the edict of making the illegal inexpressible.*

Continuing onward, App also makes an if/then distinction between the case of a lexically-scoped environment vs. anything else, and offers an additional set of match cases for both of these aforementioned cases. Both Let and Letrec make the distinction between a substitution environment vs. a dynamically- or lexically-scoped environment. Both also check to see if newly-defined variables are identical to any other previously-defined variable in its native environment; here, eval_ignore and eval_include come into play when those environments should be ignored or considered, respectively, in the substitution & evaluation. The specific details of computation are visible to the reader, although I will note that in an effort to have well-formed errors, Letrec initializes a new name' reference pointing to an unassigned variable. This, as we know it well, will appear when errors arise in variable definitions. These errors are tested against in my (quite extensive) test files test_expr.ml and test_eval.ml. Below we see evaluator in action for both eval_d and eval_l.

2.1 Implementation of evaluator

Example #1 (original)

*let x = 0. in let f = fun y -> x ** y in let x = 2. in f 6. ;;*

Example #1 in… let evaluate = eval_d ;;

```
(base) anthonymalysz@Anthonys-MacBook-Air project-2022-Awildanthony % ./miniml.byte
<== let x = 0. in let f = fun y -> x ** y in let x = 2. in f 6. ;;
==> Float(0.)
<==
```

Figure 4: *scope example function in eval_d*

Example #1 in… let evaluate = eval_l ;;

```
(base) anthonymalysz@Anthonys-MacBook-Air project-2022-Awildanthony % ./miniml.byte
<== let x = 0. in let f = fun y -> x ** y in let x = 2. in f 6. ;;
==> Float(64.)
<==
```

Figure 5: *scope example function in eval_d*

Here I've taken the opportunity to create an original example which showcases some of the extensions I've implemented aside from eval_l itself, namely the addition of floats and the power operator **. As I mentioned before, since Let and App expressions are not universal in the same sense as numbers or boolean expressions, this is what makes them so complex and unique to dynamically- and lexically-scoped environments.

Example #2 (from textbook)

*let f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 5. ;;*

Example #2 in… let evaluate = eval_d ;;

```
(base) anthonymalysz@Anthonys-MacBook-Air project-2022-Awildanthony % ./miniml.byte
<== let f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 5. ;;
==> Float(120.)
<==
```

Figure 5: *non-rec factorial function in eval_d*

Example #2 in… let evaluate = eval_l ;;

```
(base) anthonymalysz@Anthonys-MacBook-Air project-2022-Awildanthony % ./miniml.byte
<== let rec f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 5. ;;
==> Float(120.)
<==
```

Figure 6: *rec factorial function in eval_l*

Here we have the factorial (x!) function, which is by nature recursive in its mathematical definition, making it an excellent example for the purpose of demonstrating recursion in OCaml. Setting evaluate to eval_s will yield the same result as pictured in **Figure 5**. Notice that both examples evaluate to the exact same value of Float(120.); however, take special notice of the two function definitions. I specifically excluded the "rec" keyword in the definition of f in a dynamically-scoped environment, and specifically *included* it in the definition of f within the lexically-scoped environment to demonstrate the idea of scope. Without the "rec" keyword in eval_l, the function will not run because it won't recognize f. Instead, it will give the following error, an example of the well-formed error handling of an unbound variable I mentioned earlier:

```
(base) anthonymalysz@Anthonys-MacBook-Air project-2022-Awildanthony % ./miniml.byte
<== let f = fun x -> if x = 0. then 1. else x *. f (x -. 1.) in f 5. ;;
xx> evaluation error: Unbound variable f
<==
```

Figure 7: *non-rec factorial function error in eval_d*

Conclusion, Reflection, & Time Spent

This project took me about 50+ hours to complete from scratch, meaning the bare minimal code provided by the CS51 staff in order for the most essential files to compile. The trials and errors associated with writing and rewriting certain functions was an incredibly valuable experience, and makes me significantly more confident in my ability to understand what was at one point so cryptic to me in the beginning of the Spring semester. I now have a deep understanding of how a compiler works for *any* language, and I have a firm grasp on how to write a multitude of functions both efficiently and elegantly–as evidenced by my implementation of MiniML and its extensions. Overall, a relevant and interesting project.