CS51, Spring 2019
Final Project: Miniml
May 8, 2019
Authors: Seth Billiau

1 Introduction

For my CS51 final project, I have chosen to implement two main extensions to the pseudo-language miniml: support for floats and strings and lexical environment semantics. This short write-up details both of these extensions and how I have implemented them, including snapshots of code from the command line and from my .ml files.

2 Support for Floats and Strings

My first extension to miniml was the implementation of floats and strings. I will detail both of these implementations below.

The first modification I had to make to the project distribution code was a change to the expr.mli file to recognize **Float** as a tag of type float for the variant type expr. Similarly, I also modified the expr.mli file to recognize **String** as a tag of type string for the variant type *expr*. As the operations I wanted to implement for floats like negation, multiplication, etc. already had existing variant types (*binop*, *unop*, etc.) and tags (**Negate**, **Times**, etc. in the distribution code, no further edits were necessary in this file to implement floats; however, the implement strings, I added the tag **Concat** to the *binop* variant type to support the concatenation operation.

With the .mli file sorted out, I shifted my attention to the expr.ml file to implement functionality for this interface. Most of the functionality was trivial to implement as floats and strings both yield no free variables and can be substituted for themselves.

Substantive modifications to my code came when attempting to modify my binary and unary operators to support these datatypes in evaluation.ml. My current support evalution binary operators had to be extended to recognize addition, subtraction, multiplication, and comparison of floats. It also had to be altered to support concatenation of strings and raise an error if concatenation was attempted between two structures that are not both strings. Similarly the negation unary operator had to be extended to support evaluating negation of floats. My implementation of this functionality is given below:

```
let binopeval (op : binop) (v1 : expr) (v2 : expr) : expr =
  match op, v1, v2 with
  | Plus, Num x1, Num x2 -> Num (x1 + x2)
  | Plus, Float x1, Float x2 -> Float (x1 +. x2)
  | Plus, _, _ -> raise (IllFormed "Type Error")
  | Minus, Num x1, Num x2 -> Num (x1 - x2)
  | Minus, Float x1, Float x2 -> Float (x1 -. x2)
  | Minus, _, _ -> raise (IllFormed "Type Error")
  | Times, Num x1, Num x2 -> Num (x1 * x2)
  | Times, Float x1, Float x2 -> Float (x1 *. x2)
  | Times, _, _ -> raise (IllFormed "Type Error")
  | Equals, Num x1, Num x2 -> Bool (x1 = x2)
  | Equals, Bool b1, Bool b2 -> Bool (b1 = b2)
  | Equals, Float x1, Float x2 -> Bool (x1 = x2)
  | Equals, _, _ -> raise (IllFormed "Type Error")
  | LessThan, Num x1, Num x2 -> Bool (x1 < x2)
  | LessThan, Float x1, Float x2 -> Bool (x1 < x2)
  | LessThan, Bool b1, Bool b2 -> Bool (b1 < b2)
  | LessThan, _, _ -> raise (IllFormed "Type Error")
  | Concat, String s1, String s2 -> String(s1 ^ s2)
  | Concat, _, _ -> raise (IllFormed "Type Error") ;;

let unopeval (op : unop) (e : expr) : expr =
  match op, e with
  | Negate, Num x -> Num (~- x)
  | Negate, Float x -> Float (~-. x)
  | Negate, Bool b -> Bool (not b)
  | Negate, _ -> raise (IllFormed "Type Error") ;;
```

The final changes I made in the evaluation.ml file were to ensure these new data types were evaluated as "base cases" so-to-speak for my recursive evaluation functions, mirroring my initial implementation of integers.

Finally, I modified the lexical analyzer and the parser to support strings and print them in a readable way for the user of the language. The details of this implementation get tricky (adding tokens to the parser, updating the rules of tokenization), so I have decided to omit them from this write-up but the parser works quite well for these new types and is demonstrated below. I made the design decision not to alter the operators with the "." character to model OCaml as I felt that the time I would spend making this trivial altercation would be best spent testing my implementation. This demonstration uses dynamic environment semantics for evaluation.

```
<== 5. ;;
==> 5.
```

```
<== 6. + 17.5 ;;
==> 23.5
<== let x = 1. in
        let f = fun y -> x + y in
        let x = 2. in
        f 3. ;;
==> 5.
<== let rec f = fun x -> if x = 0. then ~-1. else x * f (x - 1.) in f 4. ;;
==> -24.
<== let x = 2. in let f = (fun y -> x + y) in let x = 8. in f x ;;
==> 16.
<== let x = "seth" in
        let f = (fun y -> x ^ " " ^ y) in
        let x = "maddy" in
        f "carolyn";;
==> "maddy carolyn"
<== "Stuart" ^ " " ^ "Shieber" ;;
==> "Stuart Shieber"
<== 5. + 1 ;;
Fatal error: exception Evaluation.IllFormed("Type Error")
```

3 Lexical Environment Semantics: Eval_L

I have demonstrated my implementation of dynamic environment semantics above. My implementation of lexical environment semantics is extremely similar to the implementation of eval_d in evaluation.ml. The only differences come in the match cases for the **Fun** and **App** tags in the *expr* variant type. Unlike in dynamic environment sematics, lexical semantics requires that the variables in the function be saved as a snapshot in time. This requires the use of the **Closure** tag in the *Env.value* variant type. The first implementation is shown below:

```
let rec eval_l (_exp : expr) (_env : Env.env) : Env.value =
  ...
  | App (e1, e2) ->
      (match eval_l e1 _env with
      | Closure (exp, cenv) ->
          (match exp with
          | Fun (var, e1') ->
              let newenv = extend cenv var (ref (eval_l e2 _env)) in
              eval_l e1' newenv
          | _ -> raise (EvalError "Error App: First arg not a function"))
      | _ -> raise (EvalError "Error App: First arg not a function"))
  | Fun _ -> close _exp _env ;;
```

I have demonstrated this functionality below. As is shown above, the two functions, I am using both evaluate to 5. and “maddy carolyn" above in dynamic environment semantics, but evaluate to 4. and “seth carolyn" in lexical environment semantics.

```
<== let x = 1. in
        let f = fun y -> x + y in
        let x = 2. in
        f 3. ;;
==> 4.
<== let x = "seth" in
        let f = (fun y -> x ^ " " ^ y) in
        let x = "maddy" in
        f "carolyn";;
==> "seth carolyn"
```

In my final implementation, I consolidated the functionality in dynamic and lexical evaluation into a helper function called evaluator since the two evaluation techniques are identical in all but the **App** and **Fun** tags. The helper handles the duplicitous tags and then calls functions for specific evaluation techniques as needed. This implementation is shown below.

```
(* The DYNAMICALLY-SCOPED ENVIRONMENT MODEL evaluator -- to be
   completed *)
let extract_e (v : Env.value) : expr =
  (match v with
  | Val e -> e
  | Closure (e, _) -> e) ;;

let evaluator (_exp : expr) (_env : Env.env)
    (evaltype : expr -> Env.env -> Env.value) : Env.value =
  let open Env in
  match _exp with
  | Var x -> lookup _env x
  | Num n -> Val (Num (n))
  | Float f -> Val (Float (f))
  | String s -> Val (String (s))
  | Bool b-> Val (Bool (b))
  | Unop (u, ex) -> Val(unopeval u (extract_e (evaltype ex _env)))
  | Binop (binop, e1, e2) ->
      Val(binopeval binop (extract_e (evaltype e1 _env))
                          (extract_e (evaltype e2 _env)))
  | Conditional (i, t, e) ->
      (match (extract_e (evaltype i _env)) with
      | Bool b -> if b then (evaltype t _env) else (evaltype e _env)
      | _ -> raise (EvalError "Error Cond: If is not a bool"))
  | Let (var, e1, e2) ->
     let newenv = extend _env var (ref (evaltype e1 _env)) in
     evaltype e2 newenv
  | Letrec (var, e1, e2) ->
     let unassignedref = ref (Val(Unassigned)) in
     let addunassigned = extend _env var unassignedref in
     let evaluated = evaltype e1 addunassigned in
     unassignedref := evaluated;
     evaltype e2 addunassigned
  | App _ | Fun _ -> raise (EvalError "Eval with evaltype")
  | Unassigned -> raise (EvalError "Error Unassigned: Unassigned variable")
  | Raise -> raise EvalException ;;

let rec eval_d (_exp : expr) (_env : Env.env) : Env.value =
  let open Env in
  match _exp with
  | Var _ | Float _ | String _ | Num _ | Bool _ | Unop _ | Binop _
  | Conditional _ | Let _ | Letrec _ -> evaluator _exp _env eval_d
  | App (e1, e2) ->
     (match extract_e (eval_d e1 _env) with
     | Fun (var, e1') ->
        let newenv = extend _env var (ref (eval_d e2 _env)) in
        eval_d e1' newenv
     | _ -> raise (EvalError "Error App: First arg not a function"))
  | Fun (var, e) -> Val (Fun (var, e))
  | Unassigned -> raise (EvalError "Error Unassigned: Unassigned variable")
  | Raise -> raise EvalException ;;

(* The LEXICALLY-SCOPED ENVIRONMENT MODEL evaluator -- optionally
   completed as (part of) your extension *)

let rec eval_l (_exp : expr) (_env : Env.env) : Env.value =
  let open Env in
  match _exp with
  | Var _ | Float _ | String _ | Num _ | Bool _ | Unop _ | Binop _
  | Conditional _ | Let _ | Letrec _ -> evaluator _exp _env eval_d
  | App (e1, e2) ->
     (match eval_l e1 _env with
     | Closure (exp, cenv) ->
        (match exp with
        | Fun (var, e1') ->
           let newenv = extend cenv var (ref (eval_l e2 _env)) in
           eval_l e1' newenv
        | _ -> raise (EvalError "Error App: First arg not a function"))
```

```
      | _ -> raise (EvalError "Error App: First arg not a function"))
  | Fun _ -> close _exp _env
  | Unassigned -> raise (EvalError "Error Unassigned: Unassigned variable")
  | Raise -> raise EvalException ;;
```

4 Conclusions and Next Steps:

If provided more time, I would have liked to add lists to the language. I would also have liked to expand functions to curry arguments instead of simply taking in one argument at a time. All in all, however, I'm proud of my implementation of this project.