HW3: Recursive Evaluator

CSE 4102 Project 3, Spring 2018

Bryan Arnold

3/4/2018

Section: 001

Instructor: Jeffrey A. Meunier

## Introduction

For this assignment, we wrote a program that resembles constructors from functional programming languages. This was written as an evaluator and was written in a recursive manner. Mainly, there will be an eval function that evaluates expressions and returns their values. For most expressions, the value of the expression depends on the values of the sub-expressions, which will be evaluated recursively.

## Output

### I'm going to show output for each section of the assignment to show everything works:

### Section 6.1 Output: Show env.sml still works

```
- val e1 = env_new() : int Env;

val e1 = fn : int Env

- val e2 = env_bind e1 "a" 100;

val e2 = fn : int Env

- val e3 = env_bind e2 "b" 200;

val e3 = fn : int Env

- e1 "a";

uncaught exception NameNotBound

  raised at: E:/SMLNJ/env.sml:23.41-23.53

- e2 "a";

val it = 100 : int
```

```
- e3 "a";
val it = 100 : int
- e3 "b";
val it = 200 : int
```

## Section 6.2: don’t really need output for this section

## Section 6.3: Showing the datatypes are created correctly

Note: this includes the closure datatype as well, since it was added after this section

```
- datatype Expr = Bool of bool
=               | Int of int
=               | Add of Expr * Expr
=               | If of Expr * Expr * Expr
=               | Ident of string
=               | Let of (string * Expr) list * Expr
=               | Def of string * Expr
=               | Fun of Expr * Expr
=               | App of Expr * Expr
=               | Seq of Expr list
=               | Disp of Expr
=               | Closure of Expr * Expr * Expr Env
=               | Nothing;
datatype Expr
  = Add of Expr * Expr
  | App of Expr * Expr
  | Bool of bool
  | Closure of Expr * Expr * (string -> Expr)
  | Def of string * Expr
  | Disp of Expr
  | Fun of Expr * Expr
  | Ident of string
  | If of Expr * Expr * Expr
  | Int of int
  | Let of (string * Expr) list * Expr
  | Nothing
  | Seq of Expr list
- Bool true;
val it = Bool true : Expr
- Int 100;
val it = Int 100 : Expr
- Seq [Def ("x", (Int 100)), Def ("y", (Int 200)), Add (Ident "x", Ident "y")];
val it = Seq [Def ("x",Int 100),Def ("y",Int 200),Add (Ident "x",Ident "y")]
  : Expr
- val fun1 = Fun (Ident "x", Disp (Ident "x"));
val fun1 = Fun (Ident "x",Disp (Ident "x")) : Expr
- val app1 = App (fun1, Int 100);
val app1 = App (Fun (Ident "x",Disp (Ident "x")),Int 100) : Expr
- val expr = (If ((Bool true), (Int 100), (Int 200)));
val expr = If (Bool true,Int 100,Int 200) : Expr
- val expr = (Let ([("x", Int 500), ("y", Int 50)], Add ((Ident "x"), (Ident "y"))));
val expr = Let ([("x",Int 500),("y",Int 50)],Add (Ident "x",Ident "y")) : Expr
```

This shows each datatype is defined correctly; I didn’t show closure since it is only used inside evaluating functions, so I thought it wasn’t necessary to show.

## Section 6.4: Showing each datatype can be converted to string

```
- fun e2s (Bool b) = "Bool(" ^ (Bool.toString b) ^ ")"
=   | e2s (Int i)  = "Int(" ^ (Int.toString i) ^ ")"
=   | e2s (Add (x, y)) = "Add(" ^ (e2s x) ^ "," ^ (e2s y) ^ ")"
=   | e2s (If (cond, conseq, alt)) = "If(" ^ (e2s cond) ^ "," ^ (e2s conseq) ^ "," ^ (e2s alt) ^ ")"
=   | e2s (Ident s) = "Ident(" ^ s ^ ")"
=   | e2s (Let (l, e)) = "Let([" ^ (e2s_let_helper l) ^ "]," ^ (e2s e) ^ ")"
=   | e2s (Def (s, e)) = "Def(" ^ s ^ "," ^ (e2s e) ^ ")"
=   | e2s (Fun (e, x)) = "Fun(" ^ (e2s e) ^ "," ^ (e2s x) ^ ")"
=   | e2s (App (f, p)) = "App(" ^ (e2s f) ^ "," ^ (e2s p) ^ ")"
=   | e2s (Seq s) = "Seq([" ^ (e2s_seq_helper s) ^ "])"
=   | e2s (Disp d) = "Disp(" ^ (e2s d) ^ ")"
=   | e2s (Closure (e, x, _)) = "Closure(" ^ (e2s e) ^ "," ^ (e2s x) ^ ")"
=   | e2s (Nothing) = "(null)"
=     and e2s_let_helper ([]) = ""
=       | e2s_let_helper ((id:string, e:Expr)::[]) = (id ^ ":" ^ (e2s e))
=       | e2s_let_helper ((id:string, e:Expr)::tail) =
=             (id ^ ":" ^ (e2s e) ^ "," ^ (e2s_let_helper tail))
=     and e2s_seq_helper ([]) = ""
=       | e2s_seq_helper (head::[]) = (e2s head)
=       | e2s_seq_helper (head::tail) =
=             (e2s head) ^ "," ^ (e2s_seq_helper tail);
val e2s = fn : Expr -> string
val e2s_let_helper = fn : (string * Expr) list -> string
val e2s_seq_helper = fn : Expr list -> string
-
- val expr = If (Bool true, Int 100, Int 200);
val expr = If (Bool true,Int 100,Int 200) : Expr
```

```
- e2s expr;
val it = "If(Bool(true),Int(100),Int(200))" : string
-
- val expr1 = (Add ((Int 100), (Int 200)));
val expr1 = Add (Int 100,Int 200) : Expr
- e2s expr1;
val it = "Add(Int(100),Int(200))" : string
-
- val expr2 = (Def ("x", Int 100));
val expr2 = Def ("x",Int 100) : Expr
- e2s expr2;
val it = "Def(x,Int(100))" : string
-
- val expr3 = (Let ([("x", Int 500), ("y", Int 30)], Add ((Ident "x"), (Ident "y"))));
val expr3 = Let ([("x",Int 500),("y",Int 30)],Add (Ident "x",Ident "y"))
  : Expr
- e2s expr3;
val it = "Let([x:Int(500),y:Int(30)],Add(Ident(x),Ident(y)))" : string
-
- val expr4 = (Fun (Ident "x", (Add (Ident "x", Int 50))));
val expr4 = Fun (Ident "x",Add (Ident "x",Int 50)) : Expr
- e2s expr4;
val it = "Fun(Ident(x),Add(Ident(x),Int(50)))" : string
-
- val expr5 = (App (expr4, Int 50));
val expr5 = App (Fun (Ident "x",Add (Ident "x",Int 50)),Int 50) : Expr
- e2s expr5;
val it = "App(Fun(Ident(x),Add(Ident(x),Int(50))),Int(50))" : string
```

```
-
- val expr6 = (Seq [Def ("y", Int 200), App(expr4, Ident "y")]);
val expr6 =
  Seq
    [Def ("y",Int 200),App (Fun (Ident "x",Add (Ident "x",Int 50)),Ident "y")]
  : Expr
- e2s expr6;
val it =
  "Seq([Def(y,Int(200)),App(Fun(Ident(x),Add(Ident(x),Int(50))),Ident(y))#"
  : string
- val expr7 = (Disp expr6);
val expr7 =
  Disp
    (Seq
       [Def ("y",Int 200),
        App (Fun (Ident "x",Add (Ident "x",Int 50)),Ident "y")]) : Expr
- e2s expr7;
val it =
  "Disp(Seq([Def(y,Int(200)),App(Fun(Ident(x),Add(Ident(x),Int(50))),Iden#"
  : string
- val expr7 = Nothing;
val expr7 = Nothing : Expr
- e2s expr7;
val it = "(null)" : string
```

I believe I covered each datatype e2s, if not I’ll cover them in the results of my final test cases to check if everything in this project works.

Section 6.5-6.5.2: checking if the eval function works for every expression

```
- fun eval (env:Expr Env) (Ident s) = (env, (env s))
=   | eval env (Add (x, y)) =
=     (case ((eval env x), (eval env y)) of ((_,Int c),(_,Int d)) => (env, Int (c + d))
=         | (c, d) => raise InvalidEvaluation "Add")
=   | eval env (If (cond,cons,alt)) =
=     (case (eval env cond) of
=           (_,Bool b) => if b then (eval env cons)
=                         else (eval env alt)
=         | (_) => raise InvalidCondition "Cond Requires Bool")
=   | eval env (Let (b, e)) =
=     let
=         val env2 = (eval_let_helper env b)
=         val (_,v) = (eval env2 e)
=     in
=         (env, v)
=     end
=   | eval env (Def (s, e)) = ((env_bind env s e), Nothing)
=   | eval env (Seq s) = (eval_seq_helper env s)
=   | eval env (Disp d) = ((print ((e2s d) ^ "\n")); (env, Nothing))
=   | eval env (Fun (e, x)) = (env, Closure (e, x, env))
=   | eval env (App (func, p)) =
=     (case (eval env func) of
=           (_, Closure (arg, steps, envc)) =>
=             (case arg of
=                   (Ident s) =>
=                     let
=                       val (_, param) = (eval env p)
=                       val (_, return) = (eval (env_bind envc s param) steps)
=                     in
=                       (env, return)
=                     end
=                 | (_) => raise InvalidParameter "Parameter must be a Identifier")
=         | (_) => raise InvalidApplication "Application must be applied to a Fun")
=   | eval env expr = (env, expr)
=     (* Let helper binds all of the id:value pairs in the let and returns the new
=     * resulting environment *)
=     and eval_let_helper env [] = env
=       | eval_let_helper env ((id, v)::tail) =
=         (eval_let_helper (env_bind env id v) tail)
=     (* Seq helper recursively evaluates all of the expressions in the sequence
=     * and returns the result of the final expression in the sequence *)
=     and eval_seq_helper env [] = (env, Nothing) (* edge case catch *)
=       | eval_seq_helper env (head::[]) = (eval env head)
=       | eval_seq_helper env (expr::tail) =
=         let
=           val (e,_) = (eval env expr)
=         in
=           (eval_seq_helper e tail)
=         end;
val eval = fn : Expr Env -> Expr -> Expr Env * Expr
val eval_let_helper = fn : Expr Env -> (string * Expr) list -> Expr Env
val eval_seq_helper = fn : Expr Env -> Expr list -> Expr Env * Expr
-
- val env = env_new () : Expr Env;
```

```
val env = fn : Expr Env
-
- eval env (Int 100);
val it = (fn,Int 100) : Expr Env * Expr
-
- val env1 = env_new () : Expr Env;
val env1 = fn : Expr Env
-
- eval env (Fun (Ident "x", Disp (Ident "x")));
val it = (fn,Closure (Ident "x",Disp (Ident "x"),fn)) : Expr Env * Expr
-
- val body = Add (Ident "x", Int 1);
val body = Add (Ident "x",Int 1) : Expr
-
- val abstr = Fun (Ident "x", body);
val abstr = Fun (Ident "x",Add (Ident "x",Int 1)) : Expr
-
- val app = App (abstr, Int 100);
val app = App (Fun (Ident "x",Add (Ident "x",Int 1)),Int 100) : Expr
-
- eval (env_new ()) app;
val it = (fn,Int 101) : Expr Env * Expr
```

This covers most of the datatypes being evaluated, including Nothing and Closure since they are used in only other datatypes' evaluations, but I did a bunch of my own test cases to make sure everything worked.

## Final Test Cases:

```
- val env = env_new () : Expr Env;
val env = fn : Expr Env
-
- val expr = (Add ((Int 40), (Int 20)));
val expr = Add (Int 40,Int 20) : Expr
- val disp = (Disp expr);
val disp = Disp (Add (Int 40,Int 20)) : Expr
- val (env, value) = (eval env expr);
val env = fn : Expr Env
val value = Int 60 : Expr
- val value = (Disp value);
val value = Disp (Int 60) : Expr
-
- val expr1 = (If ((Bool false), (Int 50), (Int 10)));
val expr1 = If (Bool false,Int 50,Int 10) : Expr
- val disp1 = (Disp expr1);
val disp1 = Disp (If (Bool false,Int 50,Int 10)) : Expr
- val (env, value1) = (eval env expr1);
val env = fn : Expr Env
val value1 = Int 10 : Expr
- val value1 = (Disp value1);
val value1 = Disp (Int 10) : Expr
-
- val expr3 = (Int 1000);
val expr3 = Int 1000 : Expr
- val disp3 = (Disp expr3);
```

```
val disp3 = Disp (Int 1000) : Expr
- val (env, value3) = (eval env expr3);
val env = fn : Expr Env
val value3 = Int 1000 : Expr
- val value3 = (Disp value3);
val value3 = Disp (Int 1000) : Expr
-
- val expr2 = (Bool true);
val expr2 = Bool true : Expr
- val disp2 = (Disp expr2);
val disp2 = Disp (Bool true) : Expr
- val (env, value2) = (eval env expr2);
val env = fn : Expr Env
val value2 = Bool true : Expr
- val value2 = (Disp value2);
val value2 = Disp (Bool true) : Expr
-
- val expr4 = (Def ("x", Int 42));
val expr4 = Def ("x",Int 42) : Expr
- val disp4 = (Disp expr4);
val disp4 = Disp (Def ("x",Int 42)) : Expr
- val (env, value4) = (eval env expr4);
val env = fn : Expr Env
val value4 = Nothing : Expr
- val value4 = (Disp value4);
val value4 = Disp Nothing : Expr
-
- val expr5 = (Let ([("x", Int 599), ("y", Int 2)], Add ((Ident "x"), (Ident "y"))));
```

```
val expr5 = Let ([("x",Int 599),("y",Int 2)],Add (Ident "x",Ident "y")) : Expr
- val disp5 = (Disp expr5);
val disp5 = Disp (Let ([("x",Int 599),("y",Int 2)],Add (Ident "x",Ident "y")))
  : Expr
- val (env, value5) = (eval env expr5);
val env = fn : Expr Env
val value5 = Int 601 : Expr
- val value5 = (Disp value5);
val value5 = Disp (Int 601) : Expr
-
- val expr6 = (Ident "x");
val expr6 = Ident "x" : Expr
- val disp6 = (Disp expr6);
val disp6 = Disp (Ident "x") : Expr
- val (env, value6) = (eval env expr6);
val env = fn : Expr Env
val value6 = Int 42 : Expr
- val value6 = (Disp value6);
val value6 = Disp (Int 42) : Expr
-
- val expr7 = (Fun (Ident "x", (Add (Ident "x", Int 5))));
val expr7 = Fun (Ident "x",Add (Ident "x",Int 5)) : Expr
- val disp7 = (Disp expr7);
val disp7 = Disp (Fun (Ident "x",Add (Ident "x",Int 5))) : Expr
- val (env, value7) = (eval env expr7);
val env = fn : Expr Env
val value7 = Closure (Ident "x",Add (Ident "x",Int 5),fn) : Expr
- val value7 = (Disp value7);
```

```
val value7 = Disp (Closure (Ident "x",Add (Ident "x",Int 5),fn)) : Expr
-
- val expr8 = (App (expr7, Int 300));
val expr8 = App (Fun (Ident "x",Add (Ident "x",Int 5)),Int 300) : Expr
- val disp8 = (Disp expr8);
val disp8 = Disp (App (Fun (Ident "x",Add (Ident "x",Int 5)),Int 300)) : Expr
- val (env, value8) = (eval env expr8);
val env = fn : Expr Env
val value8 = Int 305 : Expr
- val value8 = (Disp value8);
val value8 = Disp (Int 305) : Expr
-
- val expr9 = (Seq [Def ("y", Int 1), App(expr7, Ident "y")]);
val expr9 =
  Seq [Def ("y",Int 1),App (Fun (Ident "x",Add (Ident "x",Int 5)),Ident "y")]
  : Expr
- val disp9 = (Disp expr9);
val disp9 =
  Disp
    (Seq
       [Def ("y",Int 1),App (Fun (Ident "x",Add (Ident "x",Int 5)),Ident "y")])
  : Expr
- val (env, value9) = (eval env expr9);
val env = fn : Expr Env
val value9 = Int 6 : Expr
- val value9 = (Disp value9);
val value9 = Disp (Int 6) : Expr
```

Every datatype was evaluated here, so I’m confident my program works properly.

## Source Code

First, the source code for env.sml:

```
(* Environments (Homework 2), Modified for HW3 *)
(* CSE 4102 Project 2, Spring Semester, 2018 *)
(* Bryan Arnold *)
(* 3/4/2018 *)
(* Section: 001 *)
(* Instructor: Jeffrey A. Meunier *)

(* This is the declaration of the NameNotBound exception. *)
(* This exception is raised if the given search for a string *)
(* in the list of environments isn't present. *)
(* Use this the env_lookup function to check for this. *)
exception NameNotBound;

(* Type declaration for Environment type. *)
(* This is a map of a' to a' *)
(* so each environment is composed of two unknown types. *)
(* Need to use this to allow environments to be created. *)
type 'a Env = string -> 'a;

(* Function to create a new environment data structure. *)
(* This creates a new environment data strcuture that is empty.*)
(* Need to use this to allow environments to have new bindings on them. *)
fun env_new () : 'a Env = fn x => raise NameNotBound;

(* Function to assign two unknown types to a new environment. *)
(* This adds on a new mapping onto an environment so that *)
(* two unknown types can be associated with each other and fetched *)
(* Use this to add values to an individual existing environment *)
fun env_bind env name v : 'a Env = fn n => if n=name then v else (env n);

(* NOT Needed *)
(* fun env_lookup (e : Env) s : int = raise NameNotBound s; *)
```

Now, source code for eval.sml:

```
(* Bryan Arnold *)
(* CSE 4102 *)
(* HW3: Recursive Evaluator *)
(* 3/4/18 *)

(* These are used to import the functions *)
(* from env.sml, a previous assignment *)
(* with some modifications, as well as *)
(* telling the print out in the console of ML *)
(* to print out 32 lines instead of 4 when something *)
(* is too long *)
use "env.sml";
Control.Print.printDepth := 32;

(* These exceptions are used in the eval function. *)
(* The first one is raised when Add is used incorrectly, *)
(* aka when one expr isn't an integer. *)
(* The next exception is for the condition datatype, *)
(* aka when a bool value isn't given to it. *)
(* The third is for the application datatype, *)
(* aka when the parameters given are not identifiers. *)
(* The last exception is for application datatype as well, *)
(* aka when the inputs given to application are not applied *)
(* to a function expression *)
exception InvalidEvaluation of string;
exception InvalidCondition of string;
exception InvalidParameter of string;
exception InvalidApplication of string;


(**************************************************************************)


(* Expression datatype declaration/definitions. *)

(* Bool expr: this expression will function exactly the same *)
(* as the built in boolean in ML. Use this when you need a *)
(* boolean value in an expression that is to be evaluated. *)

(* Int expr: this expression will function exactly the same *)
(* as the built in integer in ML. Use this when you need an *)
(* integer value in an expression that is to be evaluated. *)

(* Add expr: this expression is to add two Int expressions together. *)
(* Use this to evaluate two Int expressions in an expression. *)
(* If expr: this is how a conditional statement would work in ML *)
(* by using If,...,then,...else,... Use this if you need a conditional *)
(* in the expression to be evaluated. *)

(*Let expr: this expression will function the same as *)
(* the let statement in ML would be. It takes a list of *)
(* string expression tuples as well as an expression as the body *)
(* use this like you would the let in ML *)

(* Def expr: this functions to allow names to be defined *)
(* in the environment. This works just like val in ML. *)
(* Use this if you need a name to be declared for an expression. *)

(* Fun expr: this expression is how an anonymous functions would *)
(* work in ML, fn x => y. Use this to define a functions just like in ML. *)

(* App expr: this expression is how one would call a function expression *)
(* previously described. Use this to call some Fun expr declared already *)

(* Seq expr: this is a list of expressions to be *)
(* evaluated. This is just like the parenthetical operator in *)
(* ML. Use this to create a sequence of expressions to be evaluated. *)

(*Disp expr: this is just a way to display any expression declared. *)
(* Use this just like the built-in display operator to display some expression. *)

(* Closure expr: this contains the environment where a *)
(* function is defined. Use this to save an environment with a *)
(* function. *)
```

```
(*Nothing expression: this is just a value to denote the null value *)
(* in ML, (). Use this to return nothing if needed. *)

datatype Expr = Bool of bool
        | Int of int
        | Add of Expr * Expr
        | If of Expr * Expr * Expr
        | Ident of string
        | Let of (string * Expr) list * Expr
        | Def of string * Expr
        | Fun of Expr * Expr
        | App of Expr * Expr
        | Seq of Expr list
        | Disp of Expr
        | Closure of Expr * Expr * Expr Env
        | Nothing;


(***************************************************************************
*********************)


(* e2s function for each expression type. *)
(* This function is responsible for taking an expression and converting it *)
(* into its string equivalent. This function can be variable on how each string is *)
(* displayed. I went with the ML constructor syntax for each expression. Use this function *)
(* to be able to properly display the expression being evaluated. *)

(* The Bool expression simply uses the toString in ML to *)
(* change the boolean expression to its string equivalent. *)

(* The Int expression simply uses the toString in ML to *)
(* change the integer expression to its string equivalent. *)

(* The Add expression displays its expression as Add(x ,y). It *)
(* displays the add as normal, while calling e2s on the two variables *)
(* that make up the add expression (2 Int expressions). *)

(* The If expression will display just as a normal conditional statement. It *)
(* won't display if,..,then,...,else, but a comma separated list of the components *)
(* of the conditional. So, expr1, expr2, expr2. *)

(* Ident expression is only a string, so displaying it will just display the string *)
(* associated with the expression. *)

(* Let expression utilizes a helper function to display, since it is a *)
(* list with an expression. It fully displays the expression string tuple list, *)
(* followed by the expression body. *)

(* Def expression is also very simple like Ident. It just has an expression string tuple, *)
(* so it displays the string, expression. *)

(* Fun expression is also simple since it is an expression tuple, *)
(* so it displays expression, expression. *)

(* App expression is the exact same as the fun expression. It displays *)
(* expression, expression. *)
```

```
(*Seq expression utilizes a helper function to display since it is a list. It *)
(* iterates through the expression list and displays each expression separated by a comma. *)

(* Disp expression is super simple, it just displays the expression associated with it. *)

(*Closure expression is a two-expression tuple, so it displays just that. *)

(*Nothing expression is nothing, so I just had it say null like other languages. *)

fun e2s (Bool b) = "Bool(" ^ (Bool.toString b) ^ ")"
  | e2s (Int i)  = "Int(" ^ (Int.toString i) ^ ")"
  | e2s (Add (x, y)) = "Add(" ^ (e2s x) ^ "," ^ (e2s y) ^ ")"
  | e2s (If (cond, conseq, alt)) = "If(" ^ (e2s cond) ^ "," ^ (e2s conseq) ^ "," ^ (e2s alt) ^ ")"
  | e2s (Ident s) = "Ident(" ^ s ^ ")"
  | e2s (Let (l, e)) = "Let([" ^ (e2s_let_helper l) ^ "]," ^ (e2s e) ^ ")"
  | e2s (Def (s, e)) = "Def(" ^ s ^ "," ^ (e2s e) ^ ")"
  | e2s (Fun (e, x)) = "Fun(" ^ (e2s e) ^ "," ^ (e2s x) ^ ")"
  | e2s (App (f, p)) = "App(" ^ (e2s f) ^ "," ^ (e2s p) ^ ")"
  | e2s (Seq s) = "Seq([" ^ (e2s_seq_helper s) ^ "])"
  | e2s (Disp d) = "Disp(" ^ (e2s d) ^ ")"
  | e2s (Closure (e, x, _)) = "Closure(" ^ (e2s e) ^ "," ^ (e2s x) ^ ")"
  | e2s (Nothing) = "(null)"
    and e2s_let_helper ([]) = ""
      | e2s_let_helper ((id:string, e:Expr)::[]) = (id ^ ":" ^ (e2s e))
      | e2s_let_helper ((id:string, e:Expr)::tail) =
            (id ^ ":" ^ (e2s e) ^ "," ^ (e2s_let_helper tail))
    and e2s_seq_helper ([]) = ""
    | e2s_seq_helper (head::[]) = (e2s head)
    | e2s_seq_helper (head::tail) =
        (e2s head) ^ "," ^ (e2s_seq_helper tail);



(*****************************************************************************
*****************************)


(* eval function for each expression type. *)
(* This function takes an expression as input, any of the ones defined *)
(* earlier and evaluates them according to what they are intended to do while placing them in an
environment. For *)
(* instance, Add will see if both expression are integers. If they are, it will *)
(* add them together, otherwise raises on exception. Use this function if you have an expression
*)
(* of any kind, and you want to evaluate it to find the value in its evaluation and place it *)
(* into the environment it was created in. *)


(*Ident expression is the simplest to evaluate. It just returns a new environment *)
(* paired with the previous environment it was in. It gets sent to env.sml to become *)
(* its own environment. *)


(* Add expression just takes two inputs, Int expression, and adds them together as *)
(* if they were integers. If one or both of the inputs isn't an Int, an exception is raised. *)


(* Cond expression evaluates the conditional expression If the condition returns true, then
evaluate conseq. *)
(* If the condition returns false, evaluate the  alt. If the condition isn't a boolean, this evaluation
*)
(* cannot be done, so return an exception. *)
```

(* Let expression utilizes a helper function to setup the environment with the bindings *)

(* given. The helper works exactly the same as the previous Let helper function, except it sets them *)

(* into an environment instead of just turning them into strings. Evaluate the expression afterwards and return the result with the original *)

(* environment given as input. *)

(* Def expression binds the definition to the environment from input and returns *)

(* the environment. Since a value has to be returned as well, but we really don't evaluate *)

(* anything from this, just return Nothing expression as the value. *)

(* Seq expression utilizes a helper function just like in the e2s function. Since Seq is *)

(* is a list, this function iterates through list, and binds each expression to the environment *)

(* and evaluates each expression. *)

(* Disp expression is responsible for turning the expression into its string *)

(* form and printing it out. Since we don't need to bind anything to an environment, *)

(* it simply just prints out the expression and return nothing as the value. *)

(* Fun expression creates and returns a closure containing the environment at the time of *)

(* function creation. Since a closure encapsulates a function, it makes sense that the *)

(* closure expression must be evaluated in this as well. *)

(* App expression finds the closure for the function binded in the environment. It then *)

(* runs the closure with the inputs given to the function. Return the value of the function with the original *)

(* environment at the end. If any components of the input are not identifiers at their lowest form, raise an exception. *)

```
(*Nothing expression doesn't need an evaluation, since it has no value. A catch all case is put *)
(* at the end just before the two helper functions. *)

fun eval (env:Expr Env) (Ident s) = (env, (env s))
  | eval env (Add (x, y)) =
    (case ((eval env x), (eval env y)) of ((_,Int c),(_,Int d)) => (env, Int (c + d))
        | (c, d) => raise InvalidEvaluation "Add")
  | eval env (If (cond,cons,alt)) =
    (case (eval env cond) of
          (_,Bool b) => if b then (eval env cons)
                        else (eval env alt)
        | (_) => raise InvalidCondition "Cond Requires Bool")
  | eval env (Let (b, e)) =
    let
        val env2 = (eval_let_helper env b)
        val (_,v) = (eval env2 e)
    in
        (env, v)
    end
  | eval env (Def (s, e)) = ((env_bind env s e), Nothing)
  | eval env (Seq s) = (eval_seq_helper env s)
  | eval env (Disp d) = ((print ((e2s d) ^ "\n")); (env, Nothing))
  | eval env (Fun (e, x)) = (env, Closure (e, x, env))
  | eval env (App (func, p)) =
    (case (eval env func) of
          (_, Closure (arg, steps, envc)) =>
            (case arg of
                  (Ident s) =>
```

```
                    let
                      val (_, param) = (eval env p)
                      val (_, return) = (eval (env_bind envc s param) steps)
                    in
                      (env, return)
                    end
                | (_) => raise InvalidParameter "Parameter must be a Identifier")
        | (_) => raise InvalidApplication "Application must be applied to a Fun")
  | eval env expr = (env, expr)
    (* Let helper binds all of the id:value pairs in the let and returns the new
    * resulting environment *)
    and eval_let_helper env [] = env
      | eval_let_helper env ((id, v)::tail) =
        (eval_let_helper (env_bind env id v) tail)
    (* Seq helper recursively evaluates all of the expressions in the sequence
    * and returns the result of the final expression in the sequence *)
    and eval_seq_helper env [] = (env, Nothing) (* edge case catch *)
      | eval_seq_helper env (head::[]) = (eval env head)
      | eval_seq_helper env (expr::tail) =
        let
          val (e,_) = (eval env expr)
        in
          (eval_seq_helper e tail)
        end;


(*******************************************************************************
**************************)
```