MP 5 – An Evaluator for PicoML

CS 421 – Fall 2017
Revision 1.1

Assigned November 17, 2017
Due November 30, 2017
Extension December 2, 2017

1 Change Log

1.1 Replaced `TrueConst` by `(BoolConst true)` and `FalseConst` by `(BoolConst false)`.

1.0 Initial Release.

2 Overview

Previously, you created a lexer, a parser, and a type inferencer for PicoML. Finally, your hard work will pay off – it is time to create an evaluator for PicoML programs. Lexing, parsing, and type inferencing will be taken care of automatically (you have already implemented these parts in previous MPs.) Your evaluator can assume that its input is correctly typed.

Your evaluator will be responsible for evaluating two kinds of things: declarations, and expressions. At top level, your evaluator will be called on a declaration or an expression with an empty memory. It will recurse on the parts, eventually returning the binding.

3 Types

For this assignment, one should note the difference between expressions and values. An expression is a syntax tree, like $2 + (4 * 3)$ or $(3 < 4)$, whereas a value is a single object, like 14 or $true$. A value is the result of evaluating an expression. Note that closures are values representing functions.

Recall that we represent PicoML programs with the following OCaml types defined in `Common`:

```
type const =
     BoolConst of bool | IntConst of int | FloatConst of float
   | StringConst of string  | NilConst | UnitConst
type bin_op = IntPlusOp | IntMinusOp | IntTimesOp | IntDivOp
            | FloatPlusOp | FloatMinusOp | FloatTimesOp | FloatDivOp
            | ConcatOp | ConsOp | CommaOp | EqOp | GreaterOp
            | ModOp | ExpoOp
type mon_op = HdOp | TlOp | PrintOp | IntNegOp | FstOp | SndOp
type exp =  (* Exceptions will be added in later MPs *)
   | VarExp of string                     (* variables *)
   | ConstExp of const                    (* constants *)
   | MonOpAppExp of mon_op * exp          (* % e1 for % is a builtin monadic operator *)
   | BinOpAppExp of bin_op * exp * exp    (* e1 % e2 for % is a builtin binary operator *)
   | IfExp of exp * exp * exp             (* if e1 then e2 else e3 *)
   | AppExp of exp * exp                  (* e1 e2 *)
   | FunExp of string * exp               (* fun x -> e1 *)
```

```
   | LetInExp of string * exp * exp      (* let x = e1 in e2 *)
   | LetRecInExp of string * string * exp * exp (* let rec f x = e1 in e2 *)
   | RaiseExp of exp                           (* raise e *)
   | TryWithExp of (exp * int option * exp * (int option * exp) list)
                                  (* try e with i -> e1 | j -> e1 | ... | k -> en *)

type dec =
     Anon of exp
   | Let of string * exp                 (* let x = exp *)
   | LetRec of string * string * exp     (* let rec f x = exp *)
```

With these, we form a PicoML abstract syntax tree. A PicoML AST will be the *input* to your evaluator. The *output* given by evaluating an AST expression has `value` type. The `value` type is defined in `Common`:

```
type memory = (string * value) list
and value =
    UnitVal                                        | BoolVal of bool
  | IntVal of int                                  | FloatVal of float
  | StringVal of string                            | PairVal of value * value
  | Closure of string * exp * memory               | ListVal of value list
  | RecVarVal of string * string * exp * memory    | Exn of int
```

Values can also be stored in memory. Memory serves as both *input* to your evaluator in general, and *output* from your evaluator when evaluating declarations. For example, one evaluates a declaration starting from some initial memory, and a list of bindings to be printed by the interpreter and an incremental memory are returned.

We will represent our memory using a `value env`. That is, we will use the `env` type from previous MPs to hold `value` types.

Recall from MP3 the use of the `'a env` type defined in `Common`:

```
type 'a env = (string * 'a) list
```

You can interact with the `env` type by using functions defined in `Common`:

```
let make_env x y = ([(x,y)]:'a env)
let lookup_env (gamma:'a env) x = lookup gamma x
let sum_env (delta:'a env) (gamma:'a env) = ((delta@gamma):'a env)
let ins_env (gamma:'a env) x y = sum_env (make_env x y) gamma


val make_env : string -> 'a -> 'a env = <fun>
val lookup_env : 'a env -> string -> 'a option = <fun>
val sum_env : 'a env -> 'a env -> 'a env = <fun>
val ins_env : 'a env -> string -> 'a -> 'a env = <fun>
```

4 Compiling, etc...

For this MP, you will only have to modify **mp5.ml** adding the functions requested. To test your code, type `make` and the three needed executables will be built: `picomlInterp`, `picomlInterpSol` and `grader`. The first two are explained below. The executable `grader` checks your implementation against the solution for a fixed set of test cases as given in the `tests` file.

4.1 Given Files

mp5.ml: This file will contain the evaluator code. This is the ONLY file that you will have to modify.

picomlInterp.ml: This file contains the main body of the `picomlInterp` and `picomlInterpSol` executables. It handles lexing, parsing, and type inferences, and calls your evaluation functions, while providing a friendly prompt to enter `PicoML` concrete syntax.

picomllex.cmo, .cmi: These files contain the compiled lexing code.

picomlyacc.cmo: This file contains the compiled parsing code.

You may want to work interactively with your code in OCaml. To facilitate your doing this, and because there are more files than usual to load, we have included in `mp5` a file `.ocamlinit` that is executed by `ocaml` every time it is started in the directory `mp5`. The contents of the file are:

```
#load "common.cmo";;
#load "picomlparse.cmo";;
#load "picomllex.cmo";;
#load "solution.cmo";;
open Common;;
#use "mp5.ml";;
```

4.2 Running PicoML

The given `Makefile` builds executables called `picomlInterp` and `picomlInterpSol`. The first is an executable for an interactive loop for the evaluator built from your solution to the assignment and the second is built from the standard solution. If you run `./picomlInterp` or `./picomlInterpSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in PicoML declarations (followed by a double semicolon), and they will be evaluated, and the resulting binding will be displayed.

At the command prompt, the programs will be evaluated (or fail evaluation) starting from the initial memory, which is empty. Each time, if evaluation is successful, the resulting memory will be displayed. Note that a program can fail at any of several stages: lexing, parsing, type inferencing, or evaluation itself. Evaluation itself will tend to fail until you have solved at least some of the problems to come.

Part 1

Problems in Part 1 of this MP are mandatory for all students. Part 2 is mandatory for only grad students. Undergrads may submit a solution for Part 2 for extra credit. Part 1 does not contain any exception handling. Part 2 will cover exceptions.

5 Problems

These problems ask you to create an evaluator for `PicoML` by writing the functions `eval_dec`, and `eval_exp` as specified. In addition, you are asked to implement the functions `const_to_val`, `monOpApply` and `binOpApply`.

For each problem, you should refer to the list of rules given as part of the problem. The rules specify how evaluation should be carried out, using natural semantics. Natural semantics were covered in class; see the lecture notes for details.

Here are some guidelines:

- `eval_dec` takes a top-level declaration and a memory, and returns a string option, value, and memory. Its type is `dec * memory -> (string option * value) * memory`.
- `eval_exp` takes an expression and a memory, and returns a value. Its type is `exp * memory -> value`.

The problems are ordered such that simpler and more fundamental concepts come first. For this reason, it is recommended that you solve the problems in the order given. Doing so may make it easier for you to test your solution before it is completed.

Here is a key to interpreting the rules:

d = top-level declaration

m = memory represented as a `value env`

e = expression

v = value

- n, i, j = integer
- r = float
- s = string
- c = monadic (unary) operator

x = identifier/variable

f = identifier/variable of functional type

t = constant

As mentioned, you should test your code in the executable PicoML environment. The problem statements that follow include some examples. However, the problem statements also contain test cases that can be used to test your implementation in the OCaml environment.

1. **Expression as a Declaration** (5 pts)

   Extend `eval_dec (dec, m)` to handle expressions that come as top-level declarations. In general (see Problem 3) `eval_dec` takes a declaration and a memory, and returns the memory updated with the bindings introduced by the declaration.

   When evaluating an expression as a declaration, since there is no concrete identifier that can be bound, we use the wildcard underscore (_) for the return identifier, represented by `None`.

$$\frac{(e, m) \Downarrow v}{(e\texttt{;;}\ , m) \Downarrow ((\_, v), m)}$$

You need to implement this rule first to be able to test other cases in the interactive top level of PicoML. We can't actually test this rule without the benefits of at least one rule for evaluating an expression.

2. **Constants** (5 pts)

   Extend `eval_exp (exp, m)` to handle non-functional constants (i.e. integers, bools, real numbers, strings, nil, and unit). For this question you will need to implement `const_to_val: const -> value`. This function takes a constant and returns the corresponding `value`.

$$\frac{}{(t, m) \Downarrow const\_to\_val(t)}$$

   In the PicoML environment,

```
> 2;;

result:
_ = 2
```

   A sample test case for the OCaml environment:

```
# eval_exp (ConstExp(IntConst 2), []);;
- : Common.value = IntVal 2
```

   The code that corresponds to what happens at the top level in `picomlInterp` is the following:

```
# eval_dec (Anon(ConstExp(IntConst 2)), []);;
- : (string option * Common.value) * Common.memory =
((None, IntVal 2), [])
```

3. **Let Declarations** (3 pts)

   Extend `eval_dec (dec, m)` to handle let-declarations. `eval_dec` takes a top-level declaration and a memory, and returns the binding introduced by the declaration together with the memory updated with that binding.

$$\frac{(e, m) \Downarrow v}{(\texttt{let}\ x = e\texttt{;;}\ , m) \Downarrow ((x, v), \{x \to v\} + m)}$$

   In the PicoML environment,

```
> let x = 2;;

result:
x = 2
```

   A sample test case for the OCaml environment:

```
# eval_dec (Let("x", ConstExp(IntConst 2)), []);;
- : (string option * Common.value) * Common.memory =
((Some "x", IntVal 2), [("x", IntVal 2)])
```

4. **Identifiers (no recursion)** (5 pts)

   Extend `eval_exp (exp, m)` to handle identifiers (i.e. variables) that are not recursive. These are identifiers in m that do not have a value of the form $RecVarVal\langle ... \rangle$, (recursive identifiers are handled later).

$$\frac{m(x) = v \quad \forall f,\ y,\ e,\ m'.\ v \neq RecVarVal(f, y, e, m')}{(x, m) \Downarrow v}$$

   Here is a sample test case.

```
# eval_exp (VarExp "x", [("x", IntVal 2)]);;
- : Common.value = IntVal 2
```

   In the PicoML environment, if you have previously successfully done Problem 5, you can test this problem with:

```
> x;;

result:
_ = 2
```

5. **Monadic Operator Application** (8 pts)

   Extend `eval_exp (exp, m)` to handle application of monadic operators `~`, `hd`, `tl`, `fst`, `snd` and `print_string`. For this question, you need to implement the function `monOpApply: mon_op -> value -> value` following the table below.

   (Hint: Check how we represent lists and pairs with the `value` type)

operator	**argument**	**operation**
`hd`	a list	return the head of the list
`tl`	a list	return the tail of the list
`fst`	a pair	return the first element of the pair
`snd`	a pair	return the second element of the pair
`~`	an integer	return the negated integer
`print_string`	a string	print the string to std_out, return unit

$$\frac{(e, m) \Downarrow v \quad monOpApply(mon, v) = v'}{(mon\ e, m) \Downarrow v'}$$

   where mon is a monadic constant function value.

   Note: Unless you are going to do Part 2, you should raise an OCaml exception if `hd` or `tl` is applied to an empty list. In Part 2, this is handled in a different way. Please see Problem 16 for the other possiblity for how to handle this.

   A sample test case in the PicoML interpreter:

```
> ~2;;

result:
_ = ~2
```

A sample test case in the OCaml environment:

```
# monOpApply IntNegOp (IntVal 2);;
- : Common.value = IntVal (-2)
# eval_exp (MonOpAppExp(IntNegOp, ConstExp (IntConst 2)), []);;
- : Common.value = IntVal (-2)
```

6. **Binary Operators** (8 pts)

Extend `eval_exp (exp, m)` to handle the application of binary operators. In the rule, we will denote the binary operator by $\oplus$. For this question, you need to implement the `binOpApply : bin_op -> (value * value) -> value` function. The table below gives the outputs for given inputs to `binOpApply`.

operator	**arguments**	**operation**
`"+"`	Two integers	Addition
`"-"`	Two integers	Subtraction
`"*"`	Two integers	Multiplication
`"/"`	Two integers	Division
`"mod"`	Two integers	Modulus
`"+."`	Two floating numbers	Addition
`"-."`	Two floating numbers	Subtraction
`"*."`	Two floating numbers	Multiplication
`"/."`	Two floating numbers	Division
`"**"`	Two floating numbers	Power
`"^"`	Two strings	Concatenation
`"::"`	A value and a list	Cons
`","`	Two values	Pairing
`"="`	Two values	Equality comparison
`">"`	Two values	Greater than

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, m) \Downarrow v_2 \quad \mathit{binOpApply}(\oplus, v_1, v_2) = v}{(e_1 \oplus e_2, m) \Downarrow v}$$

Note: For equality and other comparison operators, use the overloaded equality and comparison operators of OCaml directly on the objects of type `value`.

A sample test case.

```
# eval_exp (BinOpAppExp(IntPlusOp,
                        ConstExp(IntConst(3)),
                        ConstExp(IntConst(4))),  []) ;;
- : Common.value = IntVal 7
```

In the PicoML environment, you can test this problem with:

```
> 3 + 4;;

result:
_ = 7
```

7. **If constructs** (5 pts)

   Extend `eval_exp (exp, m)` to handle `if` constructs.

$$\frac{(e_1, m) \Downarrow true \quad (e_2, m) \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \qquad \frac{(e_1, m) \Downarrow false \quad (e_3, m) \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

   A sample test case.

```
# eval_exp (IfExp(ConstExp(BoolConst true),
                  ConstExp(IntConst 1),
                  ConstExp(IntConst 0)), []);;
- : Common.value = IntVal 1
```

   In the PicoML environment,

```
> if true then 1 else 0;;

result:
_ = 1
```

8. **Let-in expression** (6 pts)

   Extend `eval_exp (exp, m)` to handle let-in expressions.

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, \{x \rightarrow v_1\} + m) \Downarrow v_2}{(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow v_2}$$

   A sample test case.

```
# eval_exp (LetInExp ("y", ConstExp (IntConst 5), VarExp "y"), []);;
- : Common.value = IntVal 5
```

   In the PicoML environment,

```
> let y = 5 in y;;

result:
_ = 5
```

9. **Functions** (5 pts)

Extend `eval_exp (exp, m)` to handle functions. You will need to return a `ClosureVal` represented by $\langle x \to e, m \rangle$ in the rule below.

$$\frac{}{(\texttt{fun}\ x\ \texttt{->}\ e, m) \Downarrow \langle x \to e, m \rangle}$$

A sample test case.

```
# eval_exp (FunExp ("x", VarExp "x"), []);;
- : Common.value = ClosureVal ("x", VarExp "x", [])
```

In the PicoML environment,

```
> fun x -> x;;

result:
_ = <some closure>
```

10. **Function application** (6 pts)

Extend `eval_exp (exp, m)` to handle function application.

$$\frac{(e_1, m) \Downarrow \langle x \to e', m' \rangle \quad (e_2, m) \Downarrow v' \quad (e', \{x \to v'\} + m') \Downarrow v}{(e_1 e_2, m) \Downarrow v}$$

A sample test case.

```
# eval_exp (AppExp (FunExp ("x", VarExp "x"), ConstExp (IntConst 7)), []);;
- : Common.value = IntVal 7
```

In the PicoML environment,

```
> (fun x -> x) 7;;

result:
_ = 7
```

11. **Recursive Declarations** (5 pts)

Extend `eval_dec (dec, m)` to handle recursive declarations. In Picoml, recursive declarations are restricted to defining functions. Within there bodies, they are allowed to be referenced. A non-recursive function declaration evaluates to a closure containing the environment that was in effect before the function declaration was made. Variables in the body aquire their meaning from the formal parameter or from this stored environment. In the case of a recursive function declaration, we have the problem that we also may have the variable naming this function used in its body. The environment in effect before the recursive declaration does not have this variable in it. We need to create an environment that has a value for the recursive variable, but that value needs to be a closure that contains the environment we are trying to create. We solve this problem by recording a "recursive variable value" for the variable in the environment in the closure for its value instead of its actual value. This "recursive variable value" basically is a prescription for how to build the needed value, *i.e.*, the needed closure, whenever we call the variable. Since this "recursive variable value" is a prescription for building a closure, not surprisingly it will have all the components of a closure, but since it is a prescription, it has a different constructor. $RecVarVal(f, x, e, m)$

is the value we will associate with recursive function variable. It keeps track of the recursive variable f, its formal parameter x, its bound expression e, and a memory m in effect when the recursive declaration was made.

$$\frac{}{(\texttt{let rec}\ f\ x = e\ ,\ m) \Downarrow ((\texttt{Some}\ f, RecVarVal(f,x,e,m)), \{f \rightarrow RecVarVal(f,x,e,m)\} + m)}$$

A sample test case.

```
# eval_dec (LetRec ("even", "x",
 IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 0)),
  ConstExp (BoolConst true),
  IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 1)),
   ConstExp (BoolConst false),
   AppExp (VarExp "even",
    BinOpAppExp (IntMinusOp, VarExp "x", ConstExp (IntConst 2)))))), []);;

- : (string option * Common.value) * Common.memory =
((Some "even",
  RecVarVal ("even", "x",
   IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 0)),
    ConstExp (BoolConst true),
    IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 1)),
     ConstExp (BoolConst false),
     AppExp (VarExp "even",
      BinOpAppExp (IntMinusOp, VarExp "x", ConstExp (IntConst 2))))),
   [])),
 [("even",
   RecVarVal ("even", "x",
    IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 0)),
     ConstExp (BoolConst true),
     IfExp (BinOpAppExp (EqOp, VarExp "x", ConstExp (IntConst 1)),
      ConstExp (BoolConst false),
      AppExp (VarExp "even",
       BinOpAppExp (IntMinusOp, VarExp "x", ConstExp (IntConst 2))))),
    []))])
```

In the PicoML environment,

```
> let rec even x = if x = 0 then true else if x = 1 then false else even (x - 2);;

result:
even = <some recvar>
```

12. **Recursive Identifiers** (12 pts)

Extend `eval_exp (exp, m)` to handle recursive identifiers. These are identifiers that evaluate to $RecVarVal(f,x,e,m')$ for identifiers f and x, some expression e, and a memory m'.

$$\frac{m(f) = RecVarVal(g,y,e,m')}{(f,m) \Downarrow \langle y \rightarrow e, \{g \rightarrow RecVarVal(g,y,e,m')\} + m' \rangle}$$

In the PicoML environment, once you have done Problem 5, you can try:

```
> let rec even x = if x = 0 then true else if x = 1 then false else even (x - 2);;

result:
even = <some recvar>
> even 3;;

result:
_ = false
```

13. **Let-rec-in expression** (8 points)

Extend `eval_exp (exp, m)` to handle let-rec-in expressions:

$$\frac{(e_2, \{f \rightarrow RecVarVal(f, x, e_1, m)\} + m) \Downarrow v}{(\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow v}$$

In the PicoML environment, once you have done Problem 5, you can try:

```
> let rec f x = if x = 0 then 1 else x * f (x - 1) in f 3;;

result:
_ = 6
```

Part 2

This part is mandatory for grad students. It is extra credit for undergrads.

Part 1 simply ignored exceptions. In this section we include them in our language. First of all, we use the value constructor `Exn of int` in our `value` type to represent the raising of an exception.

An exception propagates through the evaluations. That is, if a subexpression of an expression evaluates to an exception, then the main expression also evaluates to the exception without evaluating the remaining subexpressions. We need to update our evaluation rules to handle this situation.

Expression Rules

Constants

$$\frac{}{(t, m) \Downarrow const\_to\_val(t)}$$

Non-recursive Variables

$$\frac{m(x) = v \quad \forall f,\ y,\ e',\ m'.\ v \neq RecVarVal(f, y, e', m')}{(x, m) \Downarrow v}$$

Monadic Operator Application

$$\frac{(e, m) \Downarrow v \quad \forall i.\ v \neq Exn(i) \quad monOpApply(mon\_op, v) = v'}{(mon\_op\ e, m) \Downarrow v'} \qquad \frac{(e, m) \Downarrow Exn(i)}{(mon\_op\ e, m) \Downarrow Exn(i)}$$

Binary Operator Application

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, m) \Downarrow v_2 \quad \forall i.\ v_1 \neq Exn(i) \wedge v_2 \neq Exn(i) \quad binOpApply(\oplus, v_1, v_2) = v}{(e_1 \oplus e_2, m) \Downarrow v}$$

$$\frac{(e_1, m) \Downarrow Exn(i)}{(e_1 \oplus e_2, m) \Downarrow Exn(i)} \qquad \frac{(e_1, m) \Downarrow v \quad \forall i.\ v \neq Exn(i) \quad (e_2, m) \Downarrow Exn(j)}{(e_1 \oplus e_2, m) \Downarrow Exn(j)}$$

If Expression

$$\frac{(e_1, m) \Downarrow true \quad (e_2, m) \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v} \qquad \frac{(e_1, m) \Downarrow false \quad (e_3, m) \Downarrow v}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow v}$$

$$\frac{(e_1, m) \Downarrow Exn(i)}{\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3 \Downarrow Exn(i)}$$

Application

$$\frac{(e_1, m) \Downarrow \langle x \to e', m' \rangle \quad (e_2, m) \Downarrow v' \quad \forall i.\ v' \neq Exn(i) \quad (e', \{x \to v'\} + m') \Downarrow v}{(e_1 e_2, m) \Downarrow v}$$

$$\frac{(e_1, m) \Downarrow Exn(i)}{(e_1 e_2, m) \Downarrow Exn(i)} \qquad \frac{(e_1, m) \Downarrow v \quad \forall j.\ v \neq Exn(j) \quad (e_2, m) \Downarrow Exn(i)}{(e_1 e_2, m) \Downarrow Exn(i)}$$

Functions

$$\frac{}{(\texttt{fun}\ x\ \texttt{->}\ e, m) \Downarrow \langle x \to e, m \rangle}$$

Let Expression

$$\frac{(e_1, m) \Downarrow v_1 \quad v_1 \neq Exn(i) \quad (e_2, m + \{x \to v_1\}) \Downarrow v_2}{(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow v_2} \qquad \frac{(e_1, m) \Downarrow Exn(i)}{(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow Exn(i)}$$

Recursive Identifiers

$$\frac{m(f) = RecVarVal(g, x, e, m')}{(f, m) \Downarrow \langle x \to e, \{g \to RecVarVal(g, x, e, m')\} + m' \rangle}$$

Declaration Rules

Expression as a Declaration

$$\frac{(e, m) \Downarrow v}{(e\texttt{;;}\ ,\ m) \Downarrow ((\texttt{None}\ , v), m)}$$

Let Declaration

$$\frac{(e, m) \Downarrow v \quad \forall i.\ v \neq Exn(i)}{(\texttt{let}\ x = e\ ,\ m) \Downarrow ((\texttt{Some}\ x, v), \{x \to v\} + m)} \qquad \frac{(e, m) \Downarrow Exn(i)}{(\texttt{let}\ x = e\ ,\ m) \Downarrow ((\texttt{None}\ , Exn(i)), m)}$$

Recursive Declarations

$$\frac{}{(\texttt{let rec}\ f\ x = e\ ,\ m) \Downarrow ((\texttt{Some}\ f, RecVarVal(f, x, e, m)), \{f \to RecVarVal(f, x, e, m)\} + m)}$$

6 Problems

14. (20 pts)

Update your implementation to incorporate exceptions in the evaluator. Follow the rules given above.

15. **Explicit exceptions** (5 pts)

Extend `eval_exp (exp, m)` to handle explicit exception raising.

$$\frac{(e,m) \Downarrow n}{(\texttt{raise}\, e, m) \Downarrow Exn(n)} \qquad \frac{(e,m) \Downarrow Exn(i)}{(\texttt{raise}\, e, m) \Downarrow Exn(i)}$$

A sample test case.

```
# eval_exp(RaiseExp (ConstExp (IntConst 1)),[]);;
- : Common.value = Exn 1
```

In the PicoML environment,

```
> raise 1;;

result:
_ = (Exn 1)
```

16. **Implicit exceptions** (4 pts)

Modify `binOpApply` and `monOpApply` to return an exception if an unexpected error occurs. In such case, $Exn(0)$ should be returned. Below are the cases you need to cover:

– An attempt to divide by zero (Both integer and real division).

– An attempt to get the head of an empty list.

– An attempt to get the tail of an empty list.

A sample test case:

```
# eval_dec (Anon
     (BinOpAppExp(IntDivOp, ConstExp (IntConst 4), ConstExp (IntConst 0))), []);;
- : (string option * Common.value) * Common.memory =
((None, Exn 0), [])
```

In the PicoML interpreter:

```
> 4/0;;

result:
val _ = (Exn 0)
```

17. **Handle expressions** (10 pts)

Extend `eval_exp (exp, m)` to handle try-with expressions.

$$\frac{(e,m) \Downarrow v \quad v \neq Exn(j)}{((\texttt{try}\ e\ \texttt{with}\ n_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ n_p\ \texttt{->}\ e_p), m) \Downarrow v}$$

$$\frac{(e,m) \Downarrow Exn(j) \qquad \forall k \leq p.(n_k \neq j \text{ and } n_k \neq \_)}{((\texttt{try}\ e\ \texttt{with}\ n_1\ \texttt{->}\ e_1\ \texttt{|}\ \ldots\ \texttt{|}\ n_p\ \texttt{->}\ e_p), m) \Downarrow Exn(j)}$$

$$\frac{(e,m) \Downarrow Exn(j) \quad (e_i,m) \Downarrow v \quad (n_i = j \text{ or } n_i = \_) \quad \forall k < i.(n_k \neq j \text{ and } n_k \neq \_)}{(\texttt{(try } e \texttt{ with } n_1 \texttt{ -> } e_1 \texttt{ | ... | } n_p \texttt{ -> } e_p), m) \Downarrow v}$$

In PicoML environment,

```
> try 4 / 0 with 0 -> 9999;;

result:
_ = 9999
```

Final Remark: Please add numerous test cases to the test suite. Try to cover obscure cases.