MP 9 – An Evaluator for MicroML

CS 421 – su 2011

Revision 1.2

Assigned July 18, 2011
Due Aug 4, 2011 23:59
Extension no extension

1 Change Log

1.3 The English description of the type of `eval_dec` was corrected. A clause limiting the second rule for application in the Part 2 to be distinct from the first rule was added, so that now all rules for application are mutually disjoint.

1.2 The example for recursive declarations was corrected so that it would type check. The corresponding test was also corrected. The rule for function application was corrected; it had the value environment update on the wrong side.

1.1 Revised points for problems, mostly upwards; corrected rule for `val _ =` e in Part 2.

1.0 Initial Release.

2 Overview

Previously, you created a lexer, a parser, and a type inferencer for MicroML. Finally, your hard work will pay off – it is time to create an evaluator for MicroML 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 MicroML programs with the following OCaml types defined in `Mp9common`:

```
(* expressions for MicroML *)

type dec =
     Val of string option * exp
   | Rec of (string * exp) nelist
   | Seq of dec * dec
   | Local of dec * dec

and exp =
   | VarExp of string
   | ConstExp of const
```

```
    | IfExp of exp * exp * exp
    | AppExp of exp * exp
    | FnExp of string pattern
    | LetExp of dec * exp
    | RaiseExp of exp
    | Handle of exp * (int option pattern nelist)

and 'a pattern = 'a * exp

type bin_op =
     IntPlusOp
   | IntMinusOp
   | IntTimesOp
   | IntDivOp
   | RealPlusOp
   | RealMinusOp
   | RealTimesOp
   | RealDivOp
   | ConcatOp
   | ConsOp
   | CommaOp
   | EqOp
   | GreaterOp

type prim_op = BinOp of bin_op | IntNeg | HdOp | TlOp | FstOp | SndOp

type const =
     BoolConst of bool
   | IntConst of int
   | RealConst of float
   | StringConst of string
   | NilConst
   | UnitConst
   | PrimOp of prim_op
```

With these, we form a MicroML abstract syntax tree. A MicroML AST will be the *input* to your evaluator. The *output* given by evaluating an AST expression is a `value` type. The `value` type is defined in `Mp9common`:

```
type value =
    UnitVal
  | BoolVal of bool
  | IntVal of int
  | RealVal of float
  | StringVal of string
  | PairVal of value * value
  | ListVal of value list
  | ClosureVal of string * exp * value env
  | PartAppBinOpVal of bin_op * value
  | RecVarVal of exp * exp env * value env
  | PrimOpVal of prim_op
  | Exn of int (* Only needed in Part 2 *)
```

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 MP5 the use of the `env` type defined in `Mp9common`:

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

You can interact with the `env` type by using functions defined in `Mp9common`:

```
val empty_env : 'a env = []
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 insert_env : 'a env -> string -> 'a -> 'a env = <fun>
```

4 Compiling, etc...

For this MP, you will only have to modify **mp9-skeleton.ml** (first convert it to **mp9.ml**), adding the functions requested. To test your code, type `make` and the three needed executables will be built: `mp9int`, `mp9intSol` and `grader`. The first two are explained below. `grader` checks your implementation against the solution for a fixed set of test cases as given in the `tests` file.

4.1 Running MicroML

The given `Makefile` builds executables called `mp9int` and `mp9intSol`. 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 `./mp9int` or `./mp9intSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in MicroML declarations (followed by double semicolons), 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 `MicroML` by writing the functions `eval_dec`, and `eval_exp` as specified. In addition, you will be asked to implement the functions `const_to_val`, `primOpApply` 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 declaration and a memory, and returns a pair of a mapping from string options to values, and memory. Its type is `dec * value env -> (string option * value) list * value env`.
- `eval_exp` takes an expression and a memory, and returns a value. Its type is `exp * value env -> 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 = declaration

m = memory stored as a `value env`

e = expression

v = value

x = identifier/variable

t = constant

b = list of bound values to be printed by the interpreter

rd = recursive definitions stored as an `exp env`

tl = tail of a list

As mentioned, you should test your code in the executable MicroML 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. **Constants** (8 pts)

   Extend `eval_exp (exp, m)` to handle constants (i.e. integers, bools, real numbers, strings, nil, unit, and primitive operators). 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 MicroML environment,

```
> 2;

result:
val it = 2
```

A sample test case for the OCaml environment:

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

2. **Val Declarations** (6 pts)

   Extend `eval_dec (dec, m)` to handle val-declarations. `eval_dec` takes a declaration and a memory, and returns a list of bindings introduced by the declaration, which will be printed by the interpreter, together with the memory containing only those bindings introduced by the declaration. If _ is encountered, we want to print out the evaluation, but not update the memory, We do this by binding the value to `None` in the list to be printed, but we do not include the binding in the memory. This is represented by the second rule.

$$\frac{(e, m) \Downarrow v}{(\texttt{val}\ x = e\ ,\ m) \Downarrow ([(\texttt{Some}\ x, v)], \{x \to v\})} \qquad \frac{(e, m) \Downarrow v}{(\texttt{val}\ \_ = e\ ,\ m) \Downarrow ([(\texttt{None}\ , v)], \{\ \})}$$

   In the MicroML environment,

```
> val x = 4;

result:
val x = 4
> val _ = 4;

result:
val _ = 4
```

   A sample test case for the OCaml environment.

```
# eval_dec (Val (Some "x", ConstExp (IntConst 4)), []);;
- : (string option * value) list * value env =
([(Some "x", IntVal 4)], [("x", IntVal 4)])
# eval_dec (Val (None, ConstExp (IntConst 4)), []);;
- : (string option * value) list * value env =
([(None, IntVal 4)], [])
```

3. **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 which are not equal to $RecVarVal\langle ...\rangle$, (recursive identifiers are handled later).

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

   Here is a sample test case.

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

In the MicroML environment, if you have previously successfully done Problem 2, you can test this problem with:

```
> x;

result:
val it = 4
```

4. **Functions** (5 pts)

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

$$\frac{}{(\texttt{fn}\ x \texttt{=>}\ e, m) \Downarrow \langle x \rightarrow e, m\rangle}$$

   A sample test case.

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

   In the MicroML environment,

```
> fn x => x;

result:
val it = <some closure>
```

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

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

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

   A sample test case.

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

   In the MicroML environment,

```
> (fn x => x) 7;

result:
val it = 7
```

6. **Primitive Operator Application** (8 pts)

   Extend `eval_exp (exp, m)` to handle application of primitive operators `~`, `hd`, `tl`, `fst`, and `snd`. For this question, you need to implement the function `primOpApply: prim_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

$$\frac{(e_1, m) \Downarrow unc \quad (e_2, m) \Downarrow v \quad primOpApply(unc, v) = v'}{(e_1 e_2, m) \Downarrow v'}$$

where unc is a unary constant function value.

Note: You should raise an OCaml exception if `hd` or `tl` is applied to an empty list.

Extend `primOpApply` to handle partial binary application (e.g. "`op` + 2"). We will denote the binary operator by $\oplus$. We represent a partial binary application using $\lceil \oplus \; v \rceil$.

$$\frac{(e_1, m) \Downarrow \texttt{op} \; \oplus \quad (e_2, m) \Downarrow v}{(e_1 e_2, m) \Downarrow \lceil \oplus \; v \rceil}$$

A sample test case in the MicroML interpreter:

```
> ~2;

result:
val it = -2
> op + 3;

result:
val it = op + 3
```

A sample test case in the OCaml environment:

```
# primOpApply IntNeg (IntVal 2);;
- : value = IntVal (-2)
```

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

   Extend `eval_exp (exp, m)` to handle the application of binary operators. 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
`"+."`	Two floating numbers	Addition
`"-."`	Two floating numbers	Subtraction
`"*."`	Two floating numbers	Multiplication
`"/."`	Two floating numbers	Division
`"^"`	Two strings	Concatenation
`"::"`	A value and a list	Cons
`","`	Two values	Pairing
`"="`	Two values	Equality comparison
`">"`	Two values	Greater than

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

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 (AppExp(AppExp(ConstExp(PrimOp(BinOp IntPlusOp)),
                       ConstExp(IntConst(3))),
                       ConstExp(IntConst(4))),  []) ;;
- : value = IntVal 7
```

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

```
> 3 + 4;

result:
val it = 7
```

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

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

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

A sample test case.

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

In the MicroML environment,

```
> if true then 1 else 0;

result:
val it = 1
```

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

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

$$\frac{(d, m) \Downarrow (b, m') \quad (e, m' + m) \Downarrow v}{(\texttt{let}\ d\ \texttt{in}\ e\ \texttt{end}\ , m) \Downarrow v}$$

   A sample test case.

```
# eval_exp (LetExp (Val (Some "y", ConstExp (IntConst 5)), VarExp "y"), []);;
- : value = IntVal 5
```

   In the MicroML environment,

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

result:
val it = 5
```

10. **Sequenced Declarations** (6 pts)

    Extend `eval_dec (dec, m)` to handle sequenced declarations.

$$\frac{(d_1, m) \Downarrow (b1, m') \quad (d_2, m' + m) \Downarrow (b2, m'')}{(d_1\ d_2, m) \Downarrow (b2 \ @\ b1, m'' + m')}$$

    A sample test case.

```
# eval_dec (Seq (Val (Some "x", ConstExp (IntConst 4)),
                 Val (Some "y", ConstExp (StringConst "hi"))), [])  ;;
- : (string option * value) list * value env =
([(Some "y", StringVal "hi"); (Some "x", IntVal 4)],
 [("y", StringVal "hi"); ("x", IntVal 4)])
```

    In the MicroML environment,

```
> val x = 4 val y = "hi";

result:
val x = 4
val y = "hi"
```

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

    Extend `eval_dec (dec, m)` to handle local declarations.

$$\frac{(d_1, m) \Downarrow (b1, m') \quad (d_2, m' + m) \Downarrow (b2, m'')}{(\texttt{local}\ d_1\ \texttt{in}\ d_2\ \texttt{end}\ , m) \Downarrow (b2, m'')}$$

    A sample test case.

```
# eval_dec (Local (Val (Some "x", ConstExp (IntConst 3)),
                   Val (Some "y", AppExp (AppExp
                   (ConstExp (PrimOp (BinOp CommaOp)),
                   VarExp "x"),
                   ConstExp (RealConst 3.14)))), []);;
- : (string option * value) list * value env =
([(Some "y", PairVal (IntVal 3, RealVal 3.14))],
 [("y", PairVal (IntVal 3, RealVal 3.14))])
```

In the MicroML environment,

```
> local val x = 3 in val y = (x, 3.14) end;

result:
val y = (3, 3.14)
```

12. **Recursive Declarations** (18 pts)

Extend `eval_dec (dec, m)` to handle recursive declarations. In order for mutually recursive identifiers to reference the expressions of other mutually recursive identifiers, we can represent a recusive variable as a `value` type. $RecVarVal(e, rd, m)$ is a recursive variable value that keeps track of its bound expression e, an `exp env` of all the recursive definitions stored as rd, and a memory m.

$$\frac{\begin{array}{c} rd = \{x_n \to e_n\} + \ldots + \{x_1 \to e_1\} \\ \forall k \leq n.\ (e_k, \{x_n \to RecVarVal(e_n, rd, m)\} + \ldots + \{x_1 \to RecVarVal(e_1, rd, m)\} + m) \Downarrow v_k \end{array}}{\begin{array}{c} (\texttt{val rec}\ x_1 = e_1\ \texttt{and}\ \ldots\ \texttt{and}\ x_n = e_n\ ,\ m) \\ \Downarrow ([(\texttt{Some}\ x_n, v_n); \ldots; (\texttt{Some}\ x_1, v_1)], \{x_n \to v_n\} + .. + \{x_1 \to v_1\}) \end{array}}$$

In the MicroML environment,

```
> val rec even = fn x => if x = 0 then true else odd (x - 1)
       and odd = fn y => if y = 0 then false else even (y - 1);

result:
val even = <some closure>
val odd = <some closure>
```

13. **Recursive identifiers** (12 pts)

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

$$\frac{\begin{array}{c} rd = \{x_n \to e_n\} + \ldots + \{x_1 \to e_1\} \\ m'' = \{x_n \to RecVarVal(e_n, rd, m')\} + \ldots + \{x_1 \to RecVarVal(e_1, rd, m')\} \\ m(x) = RecVarVal(e, rd, m') \quad (e, m'' + m') \Downarrow v \end{array}}{(x, m) \Downarrow v}$$

In the MicroML environment, once you have done Problem 12, you can try:

```
> val rec f = fn x => if x = 0 then 1 else x * f (x - 1)  val y = f 3 ;

result:
val f = <some closure>
val y = 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 evaluates. 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

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

Variables

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

If Expression

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

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

Application

$$\frac{(e_1, m) \Downarrow \mathit{Exn}(i)}{(e_1 e_2, m) \Downarrow \mathit{Exn}(i)}$$

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

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

$$\frac{(e_1, m) \Downarrow \mathit{unc} \quad (e_2, m) \Downarrow v \quad \forall i.\ v \neq \mathit{Exn}(i) \quad \mathit{UnaryApply}(\mathit{unc}, v) = v'}{(e_1 e_2, m) \Downarrow v'}$$

$$\frac{(e_1, m) \Downarrow \mathit{op}\oplus \quad (e_2, m) \Downarrow v \quad \forall i.\ v \neq \mathit{Exn}(i)}{(e_1 e_2, m) \Downarrow \lceil \oplus\ v \rceil}$$

$$\frac{(e_1, m) \Downarrow \lceil \oplus\ v_1 \rceil \quad (e_2, m) \Downarrow v_2 \quad \forall i.\ v_2 \neq \mathit{Exn}(i) \quad \mathit{BinApply}(\oplus, v_1, v_2) = v}{(e_1 e_2, m) \Downarrow v}$$

Functions

$$\frac{}{(\texttt{fn}\; x \,\texttt{=>}\, e, m) \Downarrow \langle x \to e, m\rangle}$$

Let Expression

$$\frac{(d, m) \Downarrow (b, m') \quad \forall i\; tl.\; b_1 \neq (\texttt{None}\;, Exn(i)) :: tl \quad (e, m' + m) \Downarrow v}{(\texttt{let}\; d\; \texttt{in}\; e\; \texttt{end}\;, m) \Downarrow v}$$

$$\frac{(d, m) \Downarrow ((\texttt{None}\;, Exn(i) :: tl), m')}{(\texttt{let}\; d\; \texttt{in}\; e\; \texttt{end}\;, m) \Downarrow Exn(i)}$$

Recursive Identifiers

$$\frac{\begin{array}{c} rd = \{x_n \to e_n\} + \ldots + \{x_1 \to e_1\} \\ m'' = \{x_n \to RecVarVal(e_n, rd, m')\} + \ldots + \{x_1 \to RecVarVal(e_1, rd, m')\} \\ m(x) = RecVarVal(e, rd, m') \quad (e, m'' + m') \Downarrow v \end{array}}{(x, m) \Downarrow v}$$

Declaration Rules

Val Declaration

$$\frac{(e, m) \Downarrow v \quad \forall i.\; v \neq Exn(i)}{(\texttt{val}\; x = e\;,\; m) \Downarrow ([(\texttt{Some}\; x, v)], \{x \to v\})}$$

$$\frac{(e, m) \Downarrow v}{(\texttt{val}\; \_ = e\;,\; m) \Downarrow ([(\texttt{None}\;, v)], \{\,\})}$$

$$\frac{(e, m) \Downarrow Exn(i)}{(\texttt{val}\; x = e\;,\; m) \Downarrow ([(\texttt{None}\;, Exn(i))], \{\,\})}$$

Seq Declaration

$$\frac{(d_1, m) \Downarrow (((\texttt{None}\;, Exn(i)) :: tl), m')}{(d_1 d_2, m) \Downarrow ((\texttt{None}\;, Exn(i)) :: tl, m')}$$

$$\frac{(d_1, m) \Downarrow (b1, m') \quad (d_2, m') \Downarrow (b2, m'') \quad \forall i.\; b1 \neq (\texttt{None}\;, Exn(i)) :: tl}{(d_1 d_2, m) \Downarrow (b2 \mathbin{@} b1, m'' + m')}$$

Recursive Declarations

$$\frac{\begin{array}{c} rd = \{x_n \to e_n\} + \ldots + \{x_1 \to e_1\} \\ \forall k \leq n.\ (e_k, (\{x_n \to RecVarVal(e_n, rd, m)\} + ... + \{x_1 \to RecVarVal(e_1, rd, m)\} + m) \Downarrow v_k \quad \forall i.\ v_k \neq Exn(i) \end{array}}{\begin{array}{c} (\texttt{val rec}\ x_1 = e_1\ \texttt{and}\ ...\ \texttt{and}\ x_n = e_n\ ,\ m) \\ \Downarrow ([(\texttt{Some}\ x_n, v_n); ...; (\texttt{Some}\ x_1, v_1)], \{x_n \to v_n\} + .. + \{x_1 \to v_1\}) \end{array}}$$

$$\frac{\begin{array}{c} rd = \{x_n \to e_n\} + \ldots + \{x_1 \to e_1\} \\ \forall k \leq j.\ (e_k, (\{x_n \to RecVarVal(e_n, rd, m)\} + ... + \{x_1 \to RecVarVal(e_1, rd, m)\} + m) \Downarrow v_k \\ v_j = Exn(i) \quad \forall i'.\ \forall k < j.\ v_k \neq Exn(i') \end{array}}{(\texttt{val rec}\ x_1 = e_1\ \texttt{and}\ ...\ \texttt{and}\ x_n = e_n\ ,\ m) \Downarrow ([(\texttt{None}\ , Exn(i))], \{\ \})}$$

Local Declaration

$$\frac{(d_1, m) \Downarrow (b1, m') \quad \forall i\ tl.\ b_1 \neq (\texttt{None}\ , Exn(i)) :: tl \quad (d_2, m' + m) \Downarrow (b2, m'')}{(\texttt{local}\ d_1\ \texttt{in}\ \ d_2\ \texttt{end}\ , m) \Downarrow (b2, m'')}$$

$$\frac{(d_1, m) \Downarrow ((\texttt{None}\ , Exn(i)) :: tl, m')}{(\texttt{local}\ d_1\ \texttt{in}\ \ d_2\ \texttt{end}\ , m) \Downarrow ([(\texttt{None}\ , Exn(i))], \{\ \})}$$

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.

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

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

```
> raise 1;

result:
val _ = (Exn 1)
```

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

    Modify `binOpApply` and `primOpApply` 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.

```
# eval_dec (Val (Some "it", AppExp (AppExp
      (ConstExp (PrimOp (BinOp IntDivOp)),
       ConstExp (IntConst 4)), ConstExp (IntConst 0))), []);;
- : (string option * value) list * value env =
([(None, Exn 0)], [])
```

In the MicroML interpreter:

```
> 4/0;

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

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

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

$$\frac{(e, m) \Downarrow v \quad \forall j.\ v \neq Exn(j)}{(\texttt{(handle } e \texttt{ with } n_1 \texttt{=>} e_1 \texttt{ | ... | } n_p \texttt{=>} e_p), m) \Downarrow v}$$

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

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

In MicroML environment,

```
> handle 4 / 0 with 0 => 9999;

result:
val it = 9999
```

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