
MP 8 – An Evaluator for MicroML

CS 421 – Spring 2007

Revision 1.1

Assigned April 11, 2007

Due April 18, 2007 23:59

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Correct code examples.

2 Overview

Previously, you created a lexer and a parser for MicroML. Finally, your hard work will pay off – it is time to create an evaluator. 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 and on an empty memory. It will recurse on the parts, eventually returning the final memory.

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; or it can be a closure.

Take a look at the following types from `mp8comon.ml`:

```
type exp = Unit
  | Bool of bool
  | Char of char
  | Int of int
  | Real of float
  | String of string
  | Pair of exp * exp
  | Id of string
  | LetIn of dec * exp
  | App of exp * exp
  | Fn of string * exp
  | IfThenElse of exp * exp * exp
and
dec = Valbind of (string * exp)
  | Valbindrec of (string * exp)
  | Local of (dec * dec)
  | Seq of (dec * dec)
```

These are the things that will serve as *input* to your evaluator.

Now look at the following types defined in `mp8common.ml`:

```
type memory = . . .

(*our notion of a value*)
and value = Unitval
  | Intval of int
  | Boolval of bool
  | Stringval of string
  | Realval of float
  | Charval of char
  | BuiltInOpval of string
  | Pairval of value * value
  | Listval of value list
  | Closure of string * exp * memory
  | Recvar of string * exp * memory
```

The type `value` is *output* from your evaluator when evaluating expressions. The type `memory` serves as both *input* to evaluator in general, and *output* from your evaluator when evaluating declarations and programs. For example, one evaluates a declaration starting from some initial memory, and a final memory is returned.

One interacts with memories, as we did with environments `MP5`, using the following functions, pre-defined in `mp8common.ml`:

```
(*memory operations*)
val make_mem : string -> value -> memory = <fun>
val lookup_mem : memory -> string -> value option = <fun>
val sum_mem : memory -> memory -> memory = <fun>
val ins_mem : memory -> string -> value -> memory = <fun>
val pervasive_memory : (string * value) list =
  [ ("and",
    Closure ("b1", Fn ("b2", IfThenElse (Id "b1", Id "b2", Bool false)), []));
    ("or",
    Closure ("b1", Fn ("b2", IfThenElse (Id "b1", Bool true, Id "b2")), []));
    ("not", Closure ("b", IfThenElse (Id "b", Bool false, Bool true), []));
    ("nil", Listval []); ("+", BuiltInOpval "+"); ("-", BuiltInOpval "-");
    ("*", BuiltInOpval "*"); ("/", BuiltInOpval "/"); ("<", BuiltInOpval "<");
    (">", BuiltInOpval ">"); ("<=", BuiltInOpval "<=");
    (">=", BuiltInOpval ">="); ("mod", BuiltInOpval "mod");
    ("div", BuiltInOpval "div"); ("+.", BuiltInOpval "+.");
    ("-.", BuiltInOpval "-."); ("*.", BuiltInOpval "*.");
    ("/.", BuiltInOpval "/."); ("**", BuiltInOpval "**");
    ("::", BuiltInOpval "::"); ("head", BuiltInOpval "head");
    ("tail", BuiltInOpval "tail"); ("=", BuiltInOpval "=");
    ("^", BuiltInOpval "^"); ("fst", BuiltInOpval "fst");
    ("snd", BuiltInOpval "snd")]
```

4 Compiling, etc...

For this MP, You will only have to modify `mp8-skeleton.ml` (first convert it to `mp8.ml`), adding the functions requested. To test your code, type `make` and the three needed executables will be built: `mp8int`, `mp8intSol` 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 `mp8int` and `mp8intSol`. The first is an executable for an interactive loop for the evaluator built from your solution to the assignment and the second is one built from the standard solution. If you run `./mp8int` or `./mp8intSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in MicroML declarations followed by a semicolon, and they will be evaluated, and the final memory will be displayed.

At the command prompt, the programs will be evaluated (or fail evaluation) starting from the initial memory (`Mp8common.pervasive_memory`). 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.

4.2 Given Files

mp8-skeleton.ml: This file contains the evaluator code. This is the ONLY file that you will have to modify. Change the name of the file to **mp8.ml** and work on it.

mp8int.ml: This file contains the main body of the `mp8int` and `mp8intSol` executable. It handles lexing, parsing, and type inferences, and calls your evaluation functions, while providing a friendly prompt to enter MicroML concrete syntax.

mp8lex.cmo, .cmi: These files contain the compiled lexing code.

mp8yacc.cmo: This file contains the compiled parsing code.

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 function `app_builtin`.

Modify only these functions. Do not modify any other code in `mp8.ml` or any other file. You may, however, add your own helper functions to `mp8.ml`.

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 the class; see the lecture notes for details.

Here are some guidelines:

- `eval_dec` takes a declaration and a memory, and returns a memory
- `eval_exp` takes an expression and a memory, and returns a 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 (environment)

e = expression

$v = \text{value}$

- $n = \text{integer}$
- $b = \text{bool}$
- $r = \text{real}$
- $c = \text{character}$
- $s = \text{string}$

$x = \text{identifier/variable}$

$t = \text{constant}$

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. Integers, Booleans, Real numbers, Characters, Strings and Unit (5 pts)

Extend `eval_exp e x m` to handle integers, booleans, real numbers, characters strings and unit. `eval_exp` takes an expressions and a memory, and returns the value resulting from evaluating the expression in that memory.

$$\frac{}{(n, m) \Downarrow n}$$

$$\frac{}{(b, m) \Downarrow b}$$

$$\frac{}{(r, m) \Downarrow r}$$

$$\frac{}{(c, m) \Downarrow c}$$

$$\frac{}{(s, m) \Downarrow s}$$

$$\frac{}{(() , m) \Downarrow ()}$$

2. Valbind Declaration (5 pts)

Implement `eval_dec d m`, which takes a `val` declaration and a memory and returns the memory resulting from executing the declaration. The memory returned represents exactly the bindings that were introduced by the declaration, and is an increment to the total memory accumulated by the overall program.

$$\frac{(e, m) \Downarrow v}{(\text{val } x = e, m) \Downarrow \{x \rightarrow v\}}$$

In the MicroML environment,

```
> val x = 2;
```

```
final memory:  
val x = 2;
```

A sample test case for the OCaml environemnt.

```
# eval_dec (Valbind("y", Int 2)) [];;  
- : memory = [("y", Intval 2)]
```

3. Identifiers (no recursion) (5 pts)

Extend `eval_exp ex m` to handle identifiers (i.e. variables) that are not recursive. These are identifiers in m which are not equal to `RecVar⟨...⟩`, (recursive identifiers are handled later).

$$\frac{\text{lookup_mem}(x, m) = v}{(x, m) \Downarrow v}$$

Here is a sample test case.

```
# eval_exp (Id "mod") pervasive_memory;;  
- : value = BuiltInOpval "mod"
```

4. Pairs (5 pts) Extend `eval_exp ex m` to handle pairs.

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, m) \Downarrow v_2}{((e_1, e_2), m) \Downarrow (v_1, v_2)}$$

A sample test case.

```
# eval_exp (Pair(Int 3, String "hello")) [];;  
- : value = Pairval (Intval 3, Stringval "hello")
```

5. Let-in constructs (5 pts)

Extend `eval_exp ex m` to handle let-in constructs.

$$\frac{(d_1, m) \Downarrow m' \quad (e_2, m' + m) \Downarrow v}{(\text{let } d_1 \text{ in } e_2 \text{ end}, m) \Downarrow v}$$

```
# eval_exp (LetIn (Valbind ("y", Int 5), Id "y")) [];;  
- : value = Intval 5
```

6. **Applications of Built-in Operators to primitive types** (10 pts)

Extend `eval_exp` `ex m` to handle applications of built in operators on primitive data types. Additionally, implement `app_builtin_oper v`, which takes in the operator and the operand and returns the result after calculating the value.

operator	operand	operation
" + "	Pair of integers	Addition
" - "	Pair of integers	Subtraction
" * "	Pair of integers	Multiplication
"mod"	Pair of integers	Modulus
"div"	Pair of integers	Integer division
" + . "	Pair of real numbers	Addition
" - . "	Pair of real numbers	Subtraction
" * . "	Pair of real numbers	Multiplication
" / . "	Pair of real numbers	Division
" ** "	Pair of real numbers	Power
" = "	Pair of integers	Equality comparison
" > "	Pair of integers	Greater than
" < "	Pair of integers	Less than
" >= "	Pair of integers	Greater than or equal
" <= "	Pair of integers	Less than or equal
" :: "	Pair of a value and a list	List Cons operation
" ^ "	Pair of strings	String concatenation

$$\frac{(e_1, m) \Downarrow \text{builtinop}(\text{oper}) \quad (e_2, m) \Downarrow v_1 \quad \text{app_builtin}(\text{oper}, v_1) = v}{((e_1 e_2), m) \Downarrow v}$$

A couple of sample test cases.

```
# eval_exp (App(Id "+", Pair (Int 3, Int 5))) pervasive_memory;;
- : value = Intval 8
# eval_exp (App(Id "+", Pair(Int 3,
      App(Id "-", Pair(Int 10, Int 5)))))) pervasive_memory;;
- : value = Intval 8
```

In the MicroML environment,

```
> val x = 3 * 2 - 4 * 5 + 3 mod 2 div 1;

final memory:
val x = -13;

> val x = 3.0 *. 2.0 -. 1.8 *. 4.2 /. 2.4;

final memory:
val x = 2.85;
```

7. Functions (5 pts)

Extend `eval_exp ex m` to handle functions.

You will need to return a `Closure`.

$$\frac{}{(\text{fn } x \Rightarrow e, m) \Downarrow \text{Cl}\langle x \rightarrow e, m \rangle}$$

A sample test case.

```
# eval_exp (Fn("x", Id "x")) [];;  
- : value = Closure ("x", Id "x", [])
```

In the MicroML environment,

```
> val x = fn x => x + x;
```

```
final memory:  
val x = <some closure>;
```

```
> fun f x y z = x + y + z;
```

```
final memory:  
val f = <some closure>;
```

8. Function application (5 pts)

Extend `eval_exp ex m` to handle function application.

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

A sample test case.

```
# eval_exp (App(Fn("x", Id "x"), Int 5)) [];;  
- : value = Intval 5
```

In the MicroML environment,

```
> val x = let fun f x = x + x in f 3 end;
```

```
final memory:  
val x = 6;
```

9. If constructs (5 pts)

Extend `eval_exp ex m` to handle if constructs.

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

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

```
# eval_dec (Valbind("f", IfThenElse(Bool true, Int 1, Int 0))) [];;
- : memory = [("f", Intval 1)]
```

In the MicroML environment,

```
> val x = if true then false else true;
```

```
final memory:
val x = false;
```

```
> val x = false andalso true orelse false orelse true;
```

```
final memory:
val x = true;
```

10. Local Declaration (5 pts)

Implement `eval_dec d m`, which takes a local declaration and a memory and returns the incremental memory resulting from executing the declaration.

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

A sample test case.

```
# eval_dec (Valbind("z", (LetIn(Valbind ("x", String "hi"),
    LetIn(Local(Valbind("x", Int 5),
        Valbind("y", Id "x")), Id "y"))))) pervasive_memory;;
- : memory = [("z", Intval 5)]
```

In the MicroML environment,

```
> val x = @"hi" local val x = 5 in val y = x end;
```

```
final memory:
val y = 5;
val x = "hi";
```

11. Declaration sequences (5 pts)

Implement `eval_dec d m`, which takes a sequence of declarations and a memory and returns the memory resulting from executing the declaration.

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

A sample test case.

```
# eval_exp (LetIn(Seq(Valbind("x", Int 3), Valbind("y", Int 4)),
  App(Id "+", Pair(Id "x", Id "y")))) pervasive_memory;;
- : value = Intval 7
```

In the MicroML environment,

```
> fun f x = x + x val res = f 3;
```

```
final memory:
val res = 6;
val f = <some closure>;
```

12. Recursive identifiers (15 pts)

Extend `eval_dec d m` to handle recursive identifiers (i.e. recursive variables). This rule is similar to the function rule, but returns a memory binding the recursive variable to a `Recvar` instead of a `Closure`.

$$\frac{}{(\text{val rec } f = e, m) \Downarrow \{f \rightarrow \text{RecVar}\langle f, e, m \rangle\}}$$

Also extend `eval_exp ex m` to handle recursive identifiers. These are identifiers that go to `RecVar⟨x, e, m'⟩` for some identifier x , expression e , memory m' . (You have already implemented non-recursive identifiers.)

$$\frac{\text{lookup}(x, m) = \text{RecVar}\langle f, e, m' \rangle \quad (e, \{f \rightarrow \text{RecVar}\langle f, e, m' \rangle\} + m') \Downarrow v}{(x, m) \Downarrow v}$$

A couple of test cases.

```
# eval_exp (LetIn (ValbindRec ("f", Fn ("x",
  IfThenElse (App ( Id "=", Pair (Id "x", Int 0)),
    String "finished",
    App (Id "f", App (Id "-", Pair (Id "x", Int 1))))
  )),
  App (Id "f", Int 3)) ) pervasive_memory;;
- : value = Stringval "finished"
```

```
# eval_exp (LetIn (ValbindRec ("f", Fn ("x",
  IfThenElse (App ( Id "=", Pair (Id "x", Int 0)),
    Int 1,
    App (Id "*", Pair (Id "x",
      App (Id "f", App (Id "-",
        Pair (Id "x", Int 1))))))
  )),
  App (Id "f", Int 3)) ) pervasive_memory;;
- : value = Intval 6
```

In the MicroML environment,

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

final memory:

```
val y = 6;
```

13. (Extra credit) **Application of Built-in Operators to Compound Data Types** (5 pts)

Extend each of `eval_exp ex m` and `app_builtin oper v` to handle the application of the built-in “destructor” functions `head`, `tail`, `fst`, and `snd` for the compound data types of `list` and `pair`.

operator	operand	operation
"head"	List	The head of the list
"tail"	List	The tail of the list
"fst"	Pair	The first element in the pair
"snd"	Pair	The second element in the pair