MP 8 – An Interpreter for PicoML

CS 421 – Summer 2013

Revision 1.1

Assigned July 11, 2013
Due July 22, 2013, 11:59 PM
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

1.1 Clarified the description of the return type of `eval_dec` in Problem 1.

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 interpreter for PicoML programs. Lexing, parsing, and type inferencing will be taken care of automatically (you have already implemented these parts in previous MPs.) Your interpreter can assume that its input is correctly typed.

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

3 Types

For this assignment, it is important to 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 following OCaml types:

```
(* constants for PicoML *)
type const = Bool of bool | Int of int | Float of float | String of string
           | Nil | Unit

(* Infixed binary operators for PicoML *)
type binop = Add | Sub | Mul | Div | Exp | FAdd | FSub | FMul | FDiv
           | Concat | Cons | Comma | Eq | Less

(* Primitive unary operator in PicoML *)
type monop = Head | Tail | Print | Neg | Fst | Snd

(* expressions for PicoML *)
type exp =
   | VarExp of string                 (* variables *)
   | ConExp of const                  (* constants *)
   | IfExp of exp * exp * exp         (* if exp1 then exp2 else exp3 *)
```

```
   | AppExp of exp * exp            (* exp1 exp2 *)
   | BinExp of binop * exp * exp    (* exp1 % exp2
                                       where % is a builtin binary operator *)
   | MonExp of monop * exp          (* % exp1
                                       where % is a builtin monadic operator *)
   | FunExp of string * exp         (* fun x -> exp *)
   | LetExp of string * exp * exp   (* let x = exp1 in exp2 *)
   | RecExp of string * string * exp * exp    (* let rec f x = exp1 in exp2 *)
   | RaiseExp of exp                (* raise exp *)
   | TryWithExp of exp * (int option * exp) * ((int option * exp) list)
                               (* try exp with n1 -> exp1 | ... | nm -> expm *)

(* Top-level declarations for PicoML *)
type toplvl = Anon of exp (* f 4;; *)
            | TopLet of (string * exp) (* let f = ... ;; *)
            | TopRec of (string * string * exp) (* let rec f x = ... ;; *)
```

With these, we form a PicoML abstract syntax tree. A PicoML AST will be the *input* to your interpreter. The output will use the following types defined in `Mp8common`:

```
type memory = (string * value) list
and value =
    Unitval                               | Boolval of bool
  | Intval of int                         | Floatval of float
  | Stringval of string                   | Pairval of value * value
  | Listval of value list                 | Closure of string * exp * memory
  | Recvar of string * exp * memory       | Exn of int
```

The type `value` is *output* from your interpreter when evaluating expressions. The type `memory` serves as both *input* to the interpreter in general, and *output* from the interpreter when evaluating declarations. For instance, a declaration is evaluated starting from some initial memory, and a binding and a final memory are returned.

You can interact with memories, as we did with environments in MP5, using the following functions, pre-defined in `Mp8common`:

```
(* create a new memory with the given identifier and value *)
val make_mem : string -> value -> memory = <fun>
(* look up an identifier in the given memory *)
val lookup_mem : memory -> string -> value = <fun>
(* insert a new binding to the given memory *)
val ins_mem : memory -> string -> value -> memory = <fun>
```

4 Compiling, etc.

For this MP, you will write **mp8.ml** by starting from **mp8-skeleton.ml** and adding the functions requested. To test your code, type `make` and the three needed executables will be built: `picomlint`, `picomlintSol` 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, as in previous assignments.

4.1 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.

picomlint.ml: This file contains the main body of the `picomlint` and `picomlintSol` executable. 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.

4.2 Running PicoML

The given `Makefile` builds executables called `picomlint` and `picomlintSol`. 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 `./picomlint` or `./picomlintSol`, you will get an interactive screen, much like the OCaml interactive screen. You can type in PicoML declarations (followed by double semicolons), and they will be evaluated and the resulting binding 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 in this assignment.

Part 1

Problems in Part 1 of this MP are mandatory for all students. Part 2 is extra credit, and has you add exception handling to your interpreter.

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 will be asked to implement the functions `const_to_val`, `monApply` and `binApply`. **Do not modify the code in any file other than mp8.ml.**

For each problem, you should refer to the rule(s) given as part of the problem. The rules specify how evaluation should be carried out using natural semantics, as explained in lecture.

Here are some guidelines:

- `eval_dec` takes a top-level declaration and a memory, and returns a triple of a string option, value, and memory. Its type is `toplvl * 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 (environment)

e = expression

v = value

- n, i, j = integer

- b = bool
- r = float
- s = string
- c = monadic (unary) operator

x = identifier/variable

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. `eval_dec` takes a declaration and a memory, and returns a triple of the bound identifier (if any), value of the declaration, and memory updated with the bindings introduced by the declaration.

   When evaluating an expression as a declaration, since there is no identifier to be bound, we return as the first element the wildcard underscore (_), 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. You can't actually test this rule without 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, unit). For this question you will need to implement the function `const_to_val`, which takes a constant and returns the corresponding value.

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

   In the PicoML environment,

```
> 2;;

result:
_ = 2
```

   A sample test case for the OCaml environment:

```
# eval_exp (ConExp(Int 2), []);;
- : Mp8common.value = Intval 2
```

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

```
# eval_dec (Anon(ConExp(Int 2)), []);;
- : (string option * Mp8common.value) * Mp8common.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.

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

   In the PicoML environment,

```
> let x = 2;;

result:
x = 2
```

   A sample test case for the OCaml environment:

```
# eval_dec (TopLet("x", ConExp(Int 2)), []);;
- : (string option * Mp8common.value) * Mp8common.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 $Recvar\langle ...\rangle$ (recursive identifiers are handled later).

$$\frac{m(x) = v \quad v \neq Recvar\langle y, e, m'\rangle}{(x, m) \Downarrow v}$$

   Here is a sample test case.

```
# eval_exp ((VarExp "x"), [("x", Intval 2)]);;
- : Mp8common.value = Intval 2
```

   In the PicoML environment, if you have also completed Problem 3, 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 (`monop` `hd`, `tl`, `fst`, `snd`, ~, and `print_int`. For this question, you need to implement the `monApply` function 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 result of multiply the integer by -1
`print_int`	an integer	print the integer, then return the unit value, `()`

The rule for application of monadic operators is as follows, where ‡ stands for any monadic operator:

$$\frac{(e, m) \Downarrow v_1 \quad monApply(\ddagger, v_1) \Downarrow v_2}{(\ddagger\ e, m) \Downarrow v_2}$$

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, Problem 16, this is handled in a more controlled way.

A sample test case in the PicoML interpreter:

```
> ~15;;


result:
_ = -15
```

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

   Extend `eval_exp (exp, m)` to handle the application of binary operators. We will denote binary operators by $\oplus$. For this question, you need to implement the function `binApply` . Division by 0 should raise an OCaml exception, unless you are doing Part 2.

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 floating numbers	Power
^	Two strings	Concatenation
::	A value and a list	Cons
,	Two values	Pairing
=	Two values	Equality comparison
<	Two values	Less than
>	Two values	Greater than
≤	Two values	Less than or equal
≥	Two values	Greater than or equal

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, m) \Downarrow v_2 \quad binApply(\oplus, v1, v2) = v}{((e_1\ \oplus\ e_2), m) \Downarrow v}$$

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 (BinExp (Add, ConExp(Int 3), ConExp(Int 4)), []);;
- : Mp8common.value = Intval 7
```

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

```
> 3 + 4;;

result:
_ = 7
```

7. **Let-in expressions** (5 pts)

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

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

```
# eval_exp ((LetExp("y", ConExp(Int 5), VarExp "y")), []);;
- : Mp8common.value = Intval 5
```

   In the PicoML environment,

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

result:
_ = 5
```

8. **Functions** (5 pts)

   Extend `eval_exp (exp, m)` to handle functions. You will need to return a `Closure`.

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

   A sample test case:

```
# eval_exp (FunExp("x", (BinExp (Add, VarExp "x", VarExp "x"))), []);;
- : Mp8common.value = Closure ("x", BinExp (Add, VarExp "x", VarExp "x"), [])
```

   In the PicoML environment,

```
> fun x -> x + x;;

result:
_ = <some closure>
```

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

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

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

   A sample test case:

```
# eval_exp ((AppExp(FunExp("x", VarExp "x"), ConExp(Int 5))), []);;
- : Mp8common.value = Intval 5
```

In the PicoML environment:

```
> (fun x -> x) 5 ;;

result:
_ = 5
```

10. **If expressions** (5 pts)

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

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

```
# eval_exp (IfExp(ConExp(Bool true),  ConExp(Int 1), ConExp(Int 0)), []);;
- : Mp8common.value = Intval 1
```

In the PicoML environment,

```
> if true then false else true;;

result:
_ = false
```

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

Extend `eval_dec (dec, m)` to handle recursive declarations. Recursive declarations are handled using the closure-like construct *Recvar*:

$$\frac{}{(\texttt{let rec}\ f\,x = e\texttt{;;}\ ,\ m) \Downarrow ((f, \mathit{Recvar}\langle x, e, m\rangle), m + \{f \to \mathit{Recvar}\langle x, e, m\rangle\})}$$

```
# eval_dec ((TopRec("f", "x", ConExp(Int 1))), []);;
- : (string option * Mp8common.value) * Mp8common.memory =
((Some "f", Recvar ("x", ConExp (Int 1), [])),
 [("f", Recvar ("x", ConExp (Int 1), []))])
```

In the PicoML environment, once you have done Problem 13, you can try:

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

result:
f = <some recvar>
> f 5;;

result:
_ = 120
```

12. **Let-rec-in expressions** (3 pts)

Extend `eval_exp (exp, m)` to handle let-rec bindings.

$$\frac{(e_2, m + \{f \to \mathit{Recvar}\langle x, e_1, m\rangle\}) \Downarrow v}{(\texttt{let rec}\ f\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow v}$$

In the PicoML environment,

```
> let rec f x = x + 1 in f 3;;

result:
_ = 4
```

13. **Recursive identifiers** (8 pts)

Extend `eval_exp (ex, m)` to handle recursive identifiers. These are identifiers that are mapped to $\mathit{Recvar}\langle x, e, m'\rangle$ for some expression e and memory m'. (You have already implemented non-recursive identifiers in Problem 4.)

$$\frac{m(x) = \mathit{Recvar}\langle y, e, m'\rangle}{(x, m) \Downarrow \langle y \to e, m' + \{x \to \mathit{Recvar}\langle y, e, m'\rangle\}\rangle}$$

In the PicoML environment,

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

result:
_ = 6
```

Part 2 – Extra Credit

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 interpreter. That is, if a subexpression of an expression evaluates to an exception, then the main expression also evaluates to that exception without evaluating the remaining subexpressions. We need to update our evaluation rules to handle this situation. The rules from Part 1 are updated as follows (unlisted rules stay the same):

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

$$\frac{(e, m) \Downarrow v_1 \quad v_1 \neq Exn(i) \quad monApply(\ddagger, v_1) \Downarrow v_2}{(\ddagger\ e, m) \Downarrow v}$$

$$\frac{(e_1, m) \Downarrow v_1 \quad (e_2, m) \Downarrow v_2 \quad v_1, v_2 \neq Exn(i) \quad binApply(\oplus, v1, v2) = v}{((e_1\ \oplus\ e_2), m) \Downarrow (v)}$$

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

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

Note that in the rules above we require some values to be non-exceptions. In Part 1, since we were not considering the possibility of exceptions, these premises were omitted.

Below are the rules that handle the cases where an exception may occur.

$$\frac{(e, m) \Downarrow Exn(i)}{(\texttt{let}\ x = e\texttt{;;}\ ,\ m) \Downarrow ((\_, Exn(i)), m)}$$

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

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

$$\frac{(e_1, m) \Downarrow v_1 \quad v_1 \neq Exn(j) \quad (e_2, m) \Downarrow Exn(i)}{((e_1\ \oplus\ e_2), m) \Downarrow Exn(i)}$$

$$\frac{(e_1, m) \Downarrow Exn(i)}{(\texttt{let}\ x = e_1\ \texttt{in}\ e_2, m) \Downarrow Exn(i)}$$

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

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

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

6 Problems

14. **Basic exceptions** (10 pts)

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

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

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

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

```
# eval_exp ((RaiseExp(ConExp(Int 1))), []);;
- : Mp8common.value = Exn 1
```

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

    Modify `binApply` and `monApply` 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_exp (MonExp(Head, ConExp Nil), []);;
- : Mp8common.value = Exn 0
```

In the PicoML interpreter:

```
> 4/0;;

result:
_ = (Exn 0)
```

17. **Try-with construct** (10 pts)

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

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

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

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

In the 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.