MP 10 – MiniOCaml Interpreter — Environment Model

CS 421 – Spring 2013
Revision 1.0

Assigned Thursday, April 4, 2013
Due Tuesday, April 9, 9:30am
Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objective

You will write an interpreter for MiniOCaml as you did in MP9, but using the environment model.

After completing this MP, you will have learned about the following concepts:

- How to implement functional languages using environments
- What a "closure" is

This is a short MP. Our solution is about 80 lines of code, but most of that is in the definitions of `applyOp` and `applyUnop`, which are identical what they were in MP9; in fact, there are only about 25 lines of new code.

3 Style Requirements

As in all MPs, you will be expected to meet the following style requirements. Submissions that do not meet these requirements will not receive a grade until they are resubmitted with correct styling. (Note that these requirements will be checked manually by the grader and will not be enforced by `handin`, so acceptance by `handin` does not indicate correct style.)

- No long lines. Lines are 80 characters long.
- No tabs. Use spaces for indentation.
- Indents should be no more than 4 spaces, and must be used consistently.

We will only enforce the rules just listed, but a more comprehensive style guide can be found at `http://caml.inria.fr/resources/doc/guides/guidelines.en.html`.

4 What to submit

You will submit `mp10.ml` using the `handin` program. Rename `mp10-skeleton.ml` to `mp10.ml` and start working from there.

As in previous, once you have finished appropriate sections of your interpreter, you can use the `eval` function to test individual programs, or simply add programs to the test suite.

- Download `mp10grader.tar.gz`. This tarball contains all the files you need, including the common AST file (`mp10common.ml`), the MiniOCaml lexer and parser (`miniocamllex.mll` and `miniocamlparse.mly`), and the skeleton solution file (`mp10-skeleton.ml`).
- As always, once you extract the tarball, rename `mp10-skeleton.ml` to `mp10.ml` and start modifying the file. You will modify only the `mp10.ml` file, and it is the only file that will be submitted.
- Compile your solution with `make`. Run the `./grader` to see how well you do.
- **Make sure to add several more test cases to the `tests` file; follow the pattern of the existing cases to add a new case.**
- You may use the included `testing.ml` file to run tests interactively. Open the OCaml repl and type `#use "testing.ml";;` to load all of the related modules and enable testing. Further specifics on interactive testing are included in that file.

5 Syntax and Semantics

The language you are interpreting in this MP is exactly the same as the one in MP9. See section 5 of the MP9 write-up if you want a reminder about the language.

6 Environment-based interpreter

The environment-based interpreter should give the same results as the substitution-based interpreter for all expressions, except those whose value is a function. As explained in class, here expressions are evaluated with respect to an environment giving values for variables (just as with the state in MP6).

We have provided two new definitions:

- The abstract syntax is the same as for the substitution model, but with one additional constructor:

```
  | Closure of exp * environment
```

- For the environment, we will use the same structure we did in MP 6:

```
and environment = (id * value) list
```

(Although `Closure` appears here as an abstract syntax operator, it really should not be thought of that way. It is not a simplified version of any concrete syntax structure. Rather, it is a value that appears only during evaluation of an expression. For the environment-based interpreter, it would have been "cleaner" to define a new type of "values," that included constants and closures. However, including closures in the abstract syntax is convenient in this case because it allows us to re-use `applyOp` and `applyUnop` from the substitution interpreter.)

You will need to define the following functions:

```
fetch (id:id) (env:environment) : value

extend (id:id) (v:value) (env:environment) : environment

eval (expr:exp) (env:environment) : value

applyOp (bop:binary_operation) (v1:value) (v2:value) : value

applyUnop (uop:unary_operation) (v:value) : value
```

```
type exp =
     Operation of exp * binary_operation * exp
   | UnaryOperation of unary_operation * exp
   | Var of string | StrConst of string | IntConst of int
   | True | False
   | List of exp list | Tuple of exp list
   | If of exp * exp * exp | App of exp * exp
   | Let of string * exp * exp
   | Fun of string * exp
   | Rec of string * exp
   | Closure of exp * environment

and binary_operation = Semicolon | Comma | Equals | LessThan
   | GreaterThan | NotEquals | Assign | And | Or
   | Plus | Minus | Div | Mult
   | StringAppend | ListAppend | Cons

and unary_operation = Not | Head | Tail | Fst | Snd
```

Figure 1: MiniOCaml abstract syntax

The environment works the same as in MP6, but there is one difference: In MP6, environments were always created from scratch from a list of variables and a list of values; environments were never built by extending other environments. That is because the language of MP6 had no nested scopes. MiniOCaml does, so we have added the `extend` operation, which adds a new binding to an existing environment; if the previous environment had a binding for the same name, the new binding hides it.

`applyOp` and `applyUnop` are unchanged from the substitution interpreter. If you need to see an explanation of those functions (e.g. if you didn't do MP9), consult the MP9 write-up

The definition of `eval` follows the rules in Figure 2 closely. Note that many of the rules are very similar to the ones for the substitution evaluator — just add the environment argument in the recursive calls.

(Const) Const c, $\rho \Downarrow$ Const c

(Var) a, $\rho \Downarrow \rho(a)$

(Fun) Fun(a,e), $\rho \Downarrow$ $<$Fun(a,e), $\rho >$

(Rec) Rec(f,e), $\rho \Downarrow <$ Rec(f,e), $\rho >$

(δ) e *op* e', $\rho \Downarrow v \; OP \; v'$
e, $\rho \Downarrow v$
e', $\rho \Downarrow v'$

(δ) *op* e, $\rho \Downarrow OP \; v$
e, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ True
e_2, $\rho \Downarrow v$

(If) If(e_1, e_2, e_3), $\rho \Downarrow v$
e_1, $\rho \Downarrow$ False
e_3, $\rho \Downarrow v$

(List) $[e_1, \ldots, e_n]$, $\rho \Downarrow [v_1, \ldots, v_n]$
e_1, $\rho \Downarrow v_1$
$\vdots$
e_n, $\rho \Downarrow v_n$

(Let) Let(a,e,e'), $\rho \Downarrow v'$
e, $\rho \Downarrow v$
e', $\rho[a \mapsto v] \Downarrow v'$

(App) $e \; e'$, $\rho \Downarrow v$
e, $\rho \Downarrow$ $<$Fun(a, e''), $\rho' >$
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v'] \Downarrow v$

(App) $e \; e'$, $\rho \Downarrow v''$
e, $\rho \Downarrow v$, where v = $<$Rec(f, Fun(a, e'')), $\rho' >$
e', $\rho \Downarrow v'$
e'', $\rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

Figure 2: Evaluation rules for environment model