
MP 8 – MiniOCaml Interpreter and Reducer

CS 421 – Spring 2012

Revision 1.0

Assigned Thursday, March 29, 2012

Due Tuesday, April 1, 09:30, and Thursday, April 3, 09:30

Extension 48 hours (20% penalty)

1 Change Log

1.0 Initial Release.

2 Objective

You will write an interpreter in the style of MP5, but for a simplified version of Ocaml. We will provide the lexer, parser, and translation to abstract syntax, so, as in MP 5, your task will be to interpret programs given in abstract syntax. Actually, you will write two interpreters, one using the “substitution model” and the other the “environment model”

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

- How to implement functional languages using substitution
- How to implement functional languages using environments
- What a “closure” is
- How to use higher-order functions to represent environments

This assignment is in two parts, and the due dates (and lateness policy) is a little bit complicated. The first part is the main part, where you will implement both of the interpreters mentioned above; it is due on Tuesday as usual. In the second part, you will make some fairly small modifications to the first part; those are due on Thursday. In the end, you will submit just one solution, that is, the solution to the second part; however, we will not be covering the material you need for that in class until Tuesday, so we are not making that part due until Thursday. Feel free to submit the complete solution — both parts — at any time before Tuesday.

Our solution to the first part is about 100 lines of OCaml. Much of it is similar to what you did in MP5; furthermore, the substitution-based and environment-based interpreters are very similar line-for-line, with only the function definition and application clauses being really different. The second part involves rewriting about 10 lines from the solution for the first part.

3 Collaboration

Collaboration is NOT allowed in this assignment.

4 What to submit

You will submit `mp8.ml` using the `handin` program. Rename `mp8-skeleton.ml` to `mp8.ml` and start working from there.

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

- Download `mp8grader.tar.gz`. This tarball contains all the files you need, including the common AST file (`mp8common.ml`), the MiniOCaml lexer and parser (`miniocamllex.mll` and `miniocamlparse.mly`), and the skeleton solution file (`mp8-skeleton.ml`).
- As always, once you extract the tarball, rename `mp8-skeleton.ml` to `mp8.ml` and start modifying the file. You will modify only the `mp8.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

To simplify the assignment, MiniOCaml lacks some significant features, especially pattern-matching, type definitions, and mutually-recursive functions. The concrete syntax of MiniOCaml is the same as OCaml except for those restrictions; you can see the precise syntax in the `mly` file in the tarball. The abstract syntax is given in Figure 1 (and again in `mp8common.ml` in the tarball).

In the abstract syntax in `mp8common.ml`, there are several operations that we are not asking you to implement (in your evaluators you should add a default case so that OCaml doesn't complain about not covering all the cases). These are: binary operations `;` (sequencing) and `:=` (assignment), and unary operations `ref` (create a pointer) and `!` (dereferencing, analogous to prefix `*` in C). These are the operations that implement side-effects in OCaml, something we have avoided thus far. We are *not* implementing them; they are included only so that we don't have to change the concrete and abstract syntax when we discuss them in a couple of weeks.

The syntactic form “`fun x -> e`” (which appears in both the concrete and abstract syntax) and the abstract syntax constructor `Rec`, are new. They were discussed at length in class. Also, the abstract syntax in `mp8common.ml` includes the constructor `Closure`, which is used in the environment interpreter but not in the substitution-based interpreter; see section 7.2.2.

Just as we did at first for MiniJava, you will be implementing a *dynamically-typed* version of MiniOCaml. The rules for interpreting MiniOCaml programs are given in Figure 2 (for the substitution model; see also Figure 3) and Figure 4 (for the environment model).

6 Interpreter and Reducer Structure

This week, you will be implementing the executable semantics of MiniOCaml in two different forms - an environment-based interpreter similar to what you have written before in MPs 5 and 6, plus an additional substitution-based interpreter, called a *reducer*, that directly replaces a variable with its binding upon application, rather than looking up the binding upon variable reference. The [first] interpreter's structure is quite similar to those you have written before. However, the reducer uses a new helper function, called `subst`, which implements the substitution of a variable with its binding. This function also must recursively walk the MiniOCaml AST to do its work, so you will find yourself splitting the “work” of the reducer into two separate recursively defined functions.

7 Problems

7.1 Concrete and abstract syntax

You will be dealing only with the abstract syntax in this assignment (just as in MPs 5–7); we have provided a lexer and parser for you, which translates concrete syntax to abstract syntax.

However, the translation from concrete to abstract syntax is a bit more involved than was the case for MiniJava, and it may be helpful for you to understand it. As explained in class, the abstract syntax has these properties:

- Let expressions are translated to the form “Let (*a*, *e*, *e'*),” where *a* is a variable name. A let expression that introduces a function — let *f a = e in e'* — will become, in the abstract syntax, “Let (*f*, Fun(*a*, *e*), *e'*).”
- The Rec constructor is used to indicate that a function is recursive. A recursive function definition — let rec *f a = e in e'* — becomes, in the abstract syntax, “Let (*f*, Rec(*f*, Fun(*a*, *e*)), *e'*).”
- Functions in the abstract syntax have only one argument. Functions of multiple arguments are “explicitly curried” — that is, turned into functions whose bodies are functions. Thus, let *f a b c = e in e'* becomes, in the abstract syntax, “Let (*f*, Fun(*a*, Fun(*b*, Fun(*c*, *e*))), *e'*).”

These translations all serve to simplify the evaluation process.

7.2 Part 1: The interpreters

7.2.1 Substitution-based interpreter (reducer)

The rules for evaluating expressions using substitution for function application are given in Figure 2. These rely on the definition of substitution (denoted in the evaluation rules by $e[e'/x]$), which is given in Figure 3. You need to define the following functions:

```
reduce (expr:exp) : value

subst (id:id) (v:value) (expr:exp) : exp

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

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

The type *value* is just a synonym for *exp*, but is used to indicate that the argument or result in question is in the subset of *exp*s that qualify as values, as explained in class. *applyOp* is very similar to the function of that name in MP 5, except that we have included some more operations than we did there; the arguments are expressions that represent constants — constructors *IntConst*, *StrConst*, *True*, and *False*.

To simplify the assignment, there are some changes in *applyOp*: no overloading, except for *Equals* (that is, *Plus* applies only to integers), and no short-circuit evaluation of boolean operations (so *And* and *Or* can be handled in *applyOp*). The types of each of the operators is:

```
Equals : value * value → bool
NotEquals : value * value → bool
LessThan : int * int → bool
GreaterThan : int * int → bool
And : bool * bool → bool
Or : bool * bool → bool
Plus : int * int → int
Minus : int * int → int
Div : int * int → int
Mult : int * int → int
StringAppend : string * string → string
ListAppend : value list * value list → value list
Cons : value * value list → value list
Not : bool → bool
Head : value list → value
Tail : list → value list
Fst : tuple → value
Snd : tuple → value
```

In other words, except for *Equals* and *NotEquals*, for each operations, you need only check for one type of value for each argument, and otherwise raise a type error. (To be clear, since we’re using dynamic typing, we could have overloaded operations, like *Plus*, just as in MP5. We’re not doing that just to shorten the assignment a little bit.)

7.2.2 Environment-based interpreter

The environment-based interpreter interprets the same language and should give the same result 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 MP5).

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 5:

```
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 code from the substitution interpreter, in particular `applyOp`.)

You will need to define the following functions:

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

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

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

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

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

`fetch` and `extend` should be familiar from MP 5 — `fetch` gets a value from an environment (if it is absent, throw a `TypeError`), and `extend` adds a new binding to an environment (which has precedence over any binding that variable may have had in the environment argument). `applyOp` and `applyUnop` are unchanged from the substitution interpreter.

The definition of `eval` follows the rules in Figure 4 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.

7.3 Part 2: Use functional programming in the interpreters

In this section, we just want you to modify your interpreters to use higher-order functions for a couple of specific things. Your interpreters should work exactly the same (we will only be able to tell if you’ve done this part by looking at your code). This part is intended to give you a little bit of practice with these concepts before midterm 2. In total, you will be replacing about 10 lines of code with new code.

7.3.1 Representing environment with functions

In part 1, you represented environment with a structure we have sometimes called a “dictionary.” However, an environment is really just a mapping — i.e. a function — taking identifiers to values. In OCaml, we can represent that directly:

```
and environment = id -> value
```

Your job is to rewrite `fetch` and `extend` using this new definition of environments. They have the same types (with this new environment type). (Hint: the new definitions are actually shorter than the old ones; if you are writing a lot of code for this, you are doing it wrong.)

You will also need something for the empty environment (that is, the environment represented by the dictionary “[]”). We’ll give you this:

```
let emptyEnv = fun id -> raise (TypeError ("Unbound variable: " ^ id))
```

```

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

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

7.3.2 Using map

The higher-order function `List.map` applies a function to every element in a list. In the two evaluators, there are six places where an operation is applied to every element of a list: `reduce` for lists and tuples; `eval` for lists and tuples; and `subst` for lists and tuples. If you did not use `map` (which we assume you did not), then you must have done these things by writing recursive functions to iterate over these lists. Here, you should replace those six calls to those recursive functions by calls to `List.map` with appropriate arguments; there is no need for any auxiliary function definitions (but you may need to define anonymous functions to pass to `map`).

(Const) $\text{Const } x \Downarrow \text{Const } x$	(Fun) $\text{Fun}(a, e) \Downarrow \text{Fun}(a, e)$
(Rec) $\text{Rec}(f, \text{Fun}(a, e)) \Downarrow \text{Fun}(a, e[\text{Rec}(f, \text{Fun}(a, e))/f])$	
$(\delta) e \text{ op } e' \Downarrow v \text{ OP } v'$ $e \Downarrow v$ $e' \Downarrow v'$	$(\delta) \text{op } e \Downarrow \text{OP } v$ $e \Downarrow v$
$(\text{If}) \text{If}(e_1, e_2, e_3) \Downarrow v$ $e_1 \Downarrow \text{True}$ $e_2 \Downarrow v$	$(\text{If}) \text{If}(e_1, e_2, e_3) \Downarrow v$ $e_1 \Downarrow \text{False}$ $e_3 \Downarrow v$
$(\text{List}) [e_1, \dots, e_n] \Downarrow [v_1, \dots, v_n]$ $e_1 \Downarrow v_1$ $\vdots$ $e_n \Downarrow v_n$	$(\text{App}) e \ e' \Downarrow v$ $e \Downarrow \text{Fun}(a, e'')$ $e' \Downarrow v'$ $e''[v'/a] \Downarrow v$
$(\text{Let}) \text{Let}(a, e, e') \Downarrow v'$ $e \Downarrow v$ $e'[v/a] \Downarrow v'$	

Figure 2: Evaluation rules for substitution model

For many cases, the definition of $e[e'/x]$ is just to apply it recursively, or do nothing (for constants)

$$\begin{aligned}
x[e'/x] &= e' \\
y[e'/x] &= y \\
\text{Fun}(x, e)[e'/x] &= \text{Fun}(x, e) \\
\text{Fun}(y, e)[e'/x] &= \text{Fun}(y, e[e'/x]) & (y \neq x) \\
\text{Rec}(f, e)[e'/f] &= \text{Rec}(f, e) \\
\text{Rec}(f, e)[e'/g] &= \text{Rec}(f, e[e'/g]) & (g \neq f)
\end{aligned}$$

Figure 3: Definition of substitution, $e[e'/x]$, where e is closed except possibly for occurrences of the variable being substituted for

(Const) $\text{Const } c, \rho \Downarrow \text{Const } c$	(Var) $a, \rho \Downarrow \rho(a)$
(Fun) $\text{Fun}(a, e), \rho \Downarrow \langle \text{Fun}(a, e), \rho \rangle$	(Rec) $\text{Rec}(f, e), \rho \Downarrow \langle \text{Rec}(f, e), \rho \rangle$
$(\delta) e \text{ op } e', \rho \Downarrow v \text{ OP } v'$ $\quad e, \rho \Downarrow v$ $\quad e', \rho \Downarrow v'$	$(\delta) \text{ op } e, \rho \Downarrow \text{OP } v$ $\quad e, \rho \Downarrow v$
$(\text{If}) \text{If}(e_1, e_2, e_3), \rho \Downarrow v$ $\quad e_1, \rho \Downarrow \text{True}$ $\quad e_2, \rho \Downarrow v$	$(\text{If}) \text{If}(e_1, e_2, e_3), \rho \Downarrow v$ $\quad e_1, \rho \Downarrow \text{False}$ $\quad e_3, \rho \Downarrow v$
$(\text{List}) [e_1, \dots, e_n], \rho \Downarrow [v_1, \dots, v_n]$ $\quad e_1, \rho \Downarrow v_1$ $\quad \vdots$ $\quad e_n, \rho \Downarrow v_n$	$(\text{Let}) \text{Let}(a, e, e'), \rho \Downarrow v'$ $\quad e, \rho \Downarrow v$ $\quad e', \rho[a \mapsto v] \Downarrow v'$
$(\text{App}) e \text{ } e', \rho \Downarrow v$ $\quad e, \rho \Downarrow \langle \text{Fun}(a, e''), \rho' \rangle$ $\quad e', \rho \Downarrow v'$ $\quad e'', \rho'[a \mapsto v'] \Downarrow v$	$(\text{App}) e \text{ } e', \rho \Downarrow v''$ $\quad e, \rho \Downarrow v, \quad \text{where } v = \langle \text{Rec}(f, \text{Fun}(a, e'')), \rho' \rangle$ $\quad e', \rho \Downarrow v'$ $\quad e'', \rho'[a \mapsto v', f \mapsto v] \Downarrow v''$

Figure 4: Evaluation rules for environment model