Compilers: Lab zero

Mike Spivey, September 2017

This introductory lab is entirely optional and requires very little programming, but allows you to study a simple OCaml program that incorporates a lexer and parser written with the standard tools. The materials for this lab have been incorporated in the Mercurial and Git repositories together with those for Labs 1–4.

The main aim in this lab is for you to become familiar with Objective CAML, the dialect of ML we shall be using throughout the course. You're provided with a little calculator program that lets you enter mathematical formulas and display the results (see Figure 1). Each line of input (shown here in *italic* type) is either an equation like "x = 3 + 4" or "y = x + 3", or a simple expression like "4 ∗ 5" (which the program treats as if it were an equation "it = 4 ∗ 5" with the variable "it" on the left-hand side). At present, the program will evaluate any expression that doesn't contain variables and display the answer, but it does not store the value for use in subsequent expressions. Your task is to add this feature, turning the calculator into one with an effectively unbounded set of named memories.

1 Getting started

As described in the coursebook, your first act should be to clone the Mercurial repository that contains the lab materials.

```
$ hg clone http://spivey.oriel.ox.ac.uk/hg/compilers
```

This creates a directory called compilers that contains the materials. Next, you should build some library modules that are used by all the labs.

```
$ (cd compilers/lib; make)
```

The files you need for this lab are in the directory lab0. The calculator program consists of six modules:

Files	Description
tree.mli	Defines the type *expr* of expression trees.
lexer.mll	A lexical analyser (written with *ocamllex*).
parser.mly	An expression parser (written with *ocamlyacc*).


```
$ calc
Welcome to the world of arithmetic
? x = 3 + 4
=> 7.0
? 4 * 5
=> 20.0
? y = x - 1.5
Failure: sorry, I don't do variables
? ^D
Bye
$
```

Figure 1: *Using the calculator*

memory.mli, memory.ml	Memory for values of variables.
eval.mli, eval.ml	Functions for evaluating expressions
main.ml	The main program.

Your task will involve modifying the *Eval* module, and adding an implementation of the module *Memory* that holds the contents of the calculator's memory.

The practical kit includes a makefile that describes how to build the complete calculator program: just change to the directory compilers/lab0 and give the command make:

```
$ cd compilers/lab0
$ make
```

The following steps will be executed automatically:

```
ocamllex lexer.mll
16 states, 429 transitions, table size 1812 bytes
ocamlyacc parser.mly
ocamlc -c  tree.mli
ocamlc -c  parser.mli
ocamlc -c  lexer.mli
ocamlc -c  lexer.ml
ocamlc -c  parser.ml
ocamlc -c  eval.mli
ocamlc -c  eval.ml
ocamlc -c  main.ml
ocamlc ../lib/common.cma
   lexer.cmo parser.cmo eval.cmo main.cmo -o calc
```

Two of these commands invoke the tools *ocamllex* and *ocamlyacc* to generate ML code the for lexer and parser; the rest use the Objective CAML compiler on various files with names like module.mli and module.ml that contain the interface and the implementation of each module in the program. From the interface file of a module, the compiler produces a file module.cmi that contains the interface in binary form; when the compiler subsequently processes the corresponding implementation file, or another module that uses this one,

it consults this file to check that the definition or use of functions agrees with the declared interface.

Compiling an implementation file produces a file called module.cmo that contains object code for the module: this file is linked together with others to build the final program. The Objective CAML compiler as we shall use it does not produce object code for a real machine, but for an invented machine, rather like the virtual machines we shall be using in the course. That's why the files of object code have the extension .cmo, rather than the extension .o that is used for files of genuine object code.

As the example shows, it's possible for a module to have no interface file: there is no file main.mli to go with the file main.ml. The convention is that this means *all* the functions defined in those modules are accessible to others; although actually the module *main* is used here for its side-effect alone: it contains the main program. Compiling a '.ml' file that doesn't have a corresponding '.mli' file produces *both* a '.cmi' file that contains the interface and a '.cmo' file that contains the object code. It's also possible to have a module with an interface but no implementation: the file tree.mli just defines the type of expression trees, and leaves nothing to be implemented.

The final command puts together all the files of object code, and the library archive ../lib/common.cma, to produce an executable program *calc* that you can run just like any other. When you type calc at the shell prompt,[1] what actually happens behind the scenes is that an interpreter for the virtual machine code is started, and it is given the code that the compiler generated for your program. This layer of interpretation means that Objective CAML programs go about 5 times slower than they would if they were really translated into proper machine code, but I promise you won't notice the difference, at least with the small test cases we'll use in our lab sessions.[2]

2 A guided tour

The lexer and parser for expressions are built using the program generators *ocamllex* and *ocamlyacc* that we shall shortly be looking at more closely; for now, you can just take for granted the job that they do. The main program contains a loop that reads lines from the keyboard, and uses the lexer and parser to build for each expression a tree of the type *expr* that is defined in the file tree.mli:

```
type expr =
    Number of float                (* Constant (value) *)
  | Variable of string             (* Variable (name) *)
  | Binop of op * expr * expr      (* Binary operator *)

and op = Plus | Minus | Times | Divide
```

The type *float* is Objective CAML's representation of real numbers: I've used

[1] Don't forget to type ./calc instead if your account is set up so that the path does not contain the current directory

[2] For critical, production applications, there is a compiler that can translate Objective CAML into real machine code for a number of different machines. Since this compiler is itself much slower than the one that generates virtual machine code – and since we'll spend much more time building our compilers than running them – it makes no sense for us to use it.

```
open Tree

(* do_binop – compute result of binary operator *)
let do_binop w v1 v2 =
  match w with
      Plus → v1 +. v2
    | Minus → v1 −. v2
    | Times → v1 *. v2
    | Divide →
        if v2 = 0.0 then failwith "dividing by zero";
        v1 /. v2

(* eval_expr – evaluate an expression *)
let rec eval_expr =
  function
      Number r → r
    | Variable x → failwith "Sorry, I don't do variables"
    | Binop (w, e1, e2) →
        do_binop w (eval_expr e1) (eval_expr e2)

(* process – process an equation, return value of RHS *)
let process (x, e) = eval_expr e
```

Figure 2: *The file* eval.ml

them in place of integers because doing so makes the calculator program actually useful. We won't always be bothered by such considerations later in the course!

As an example, if the input were the string 3+4*(5+6), then the following tree of type *expr* would be created:

```
Binop (Plus,
        Number 3.0,
        Binop (Times,
                Number 4.0,
                Binop (Plus, Number 5.0, Number 6.0)))
```

Each line from the input can actually consist of an equation with a variable on the left and an expression on the right. The main program calls the function *process* from the *Eval* module, passing as arguments the identifier (or "it" if none was given) and the tree for the expression. The type of *process* is specified in eval.mli; here is the whole of that file:

```
(* process – process an equation, return value of RHS *)
val process : string * Tree.expr → float
```

The file eval.ml is shown in Figure 2. The function *do_binop* of type

$$do\_binop : op \to float \to float \to float$$

combines two floating-point numbers using a binary operation; the operators written as +., −., etc., are Objective CAML's way of doing floating-point arithmetic. The function *eval_expr* with type

$$eval\_expr : expr \to float$$

evaluates an expression by using *do_binop* to perform each operation, and calls itself recursively to evaluate sub-expressions. This initial version does not handle variables.

What *process* (*x*, *e*) should do is to evaluate the right-hand side *e* of the equation *x* = *e* that has been input, and store the value against the identifier *x* on the left-hand side, also returning the value so that the main program can print it. However, the initial version of *process* simply evaluates *e* and returns the value without storing it anywhere.

3 A memory module

Your first job will be to build a module for the calculator's memory. An interface file memory.mli already exists:

```
(* store – set a memory (named by a string) to a given value *)
val store : string → float → unit

(* recall – retrieve the value from a given memory, or fail *)
val recall : string → float
```

You should write the implementation file memory.ml for this module. You can use any method you choose. The simplest method is to use a list of (variable, value) pairs and the standard function *assoc* from the *List* module; another method is to use a list of records. You also might like to investigate the *Hashtbl* and *Map* modules from the library; the first provides a generic implementation of hash tables that you could use, and the second provides a generic implementation of finite mappings as ordered trees. Any of these methods gives an implementation of *Memory* in only a few lines. The functionality is really too simple to be worth putting in a module, except as an illustration of the module mechanism.

Having written this implementation, you should add the *Memory* module to the calculator by replacing the line in *Makefile* that says

```
CALC = lexer.cmo parser.cmo eval.cmo main.cmo
```

with the following line:

```
CALC = lexer.cmo parser.cmo memory.cmo eval.cmo main.cmo
```

It's important to list the modules in the order shown, so that each module is loaded before the ones that use it.

4 Extending the evaluator

There are a couple of places where you'll need to make small extensions to the evaluator in eval.ml. First, you should add a line near the beginning that opens the *Memory* module:

```
open Memory
```

This line makes it possible to refer to the *store* and *recall* functions directly: it's not absolutely necessary, because you could achieve the same effect by writing *Memory.store* and *Memory.recall* in place of *store* and *recall*.

```
$ calc
Welcome to the world of arithmetic
? x = 3 + 4
=> 7.0
? 4 * 5
=> 20.0
? y = x - 1.5
=> 5.5
? ^D
Bye
$
```

Figure 3: *The extended calculator*

Next, remove the call to *failwith* from *eval_expr*, and replace it with something appropriate. Finally, change *process* so that it saves the value of the expression before returning it.

With these changes, you should be able to rebuild the calculator and get the results shown in Figure 3.