Chapter 5

Interlude: μScheme in ML

Contents

The interpreters in Chapters 2 through 4 are written in C. C is a very good language in which to write a garbage collector: it is simple and widely used, and it provides the low-level, unsafe features that we need. For illustrating programming languages, however, C is far from ideal. In this and succeeding chapters, we therefore present interpreters written in the functional language Standard ML. Standard ML is particularly well suited to symbolic computing, and the connection between language design, formal semantics, and implementations is much more clearly illustrated by an ML program than by a C program. As an introduction to writing interpreters in ML, and to show the continuity between the first and second halves of this book, we present a μScheme interpreter written in ML.

If you happen to be a seasoned ML programmer, you may find our code a bit strange, because we assume you have never seen Standard ML modules. Our code is not broken into modules; it is packed into a single source file. Avoiding modules is poor style, and it makes it impossible to take full advantage of the libraries that come with many implementations of ML, including Standard ML of New Jersey and Moscow ML. But by avoiding modules, we ease the transition for readers whose only previous experience with functional languages is their work with μScheme in Chapter 3.

For CS45600, Purdue University, Spring 2014 only --- do nc

Because we don't use ML modules, we have no formal way to talk about interfaces and to distinguish interfaces from implementations. We work around this problem using a literate-programming trick: we put the types of functions and values, which is mostly what ML interfaces describe, in boxes preceding the implementations. This technique makes it possible to present an interface formally just before we present its implementation. The Noweb processor ensures that the material in the boxes is checked by the ML compiler.

The interpreter in this chapter has the same structure as the interpreter in Chapter 3; as before, we have environments, abstract syntax, an evaluator for expressions, primitives, and an evaluator for definitions. The details of initialization and creating a program are somewhat different in ML than in C, and we use the spelling and capitalization conventions recommended by the SML'97 Standard Basis Library (Gansner and Reppy 2002).

5.1 Environments

In our C implementation, an environment binds names to locations. In our ML implementation, an environment binds names to mutable cells, but that's not all. We actually provide a *polymorphic* implementation in which an environment of type `'a env` binds names to values of type `'a`. The *type variable* `'a` stands for an unknown type. A type variable can be *instantiated* at any type; to get an environment binding names to mutable cells, we instantiate our implementation using `'a` = `value ref`.

We represent an environment of type `'a env` as a list of (`name`, `'a`) pairs. The declarations in the box give the interface to our implementation; through some Noweb hackery, they are checked by the ML compiler.

214 ⟨*environments* 214⟩≡ (224)

```
emptyEnv : 'a env
find     : name * 'a env -> 'a
bind     : name      * 'a      * 'a env -> 'a env
bindList : name list * 'a list * 'a env -> 'a env
```

```
type name = string
type 'a env = (name * 'a) list
val emptyEnv = []

(* lookup and assignment of existing bindings *)
exception NotFound of name
fun find (name, []) = raise NotFound name
  | find (name, (n, v)::tail) = if name = n then v else find (name, tail)

(* adding new bindings *)
exception BindListLength
fun bind (name, v, rho) = (name, v) :: rho
fun bindList (n::vars, v::vals, rho) = bindList (vars, vals, bind (n, v, rho))
  | bindList ([], [], rho) = rho
  | bindList _ = raise BindListLength
```

Because ML strings are immutable, we can use them to represent names directly. We also use exceptions, not an `error` procedure, to indicate when things have gone wrong. The exceptions we use are listed in Table 5.1.

For CS45600, Purdue University, Spring 2014 only --- do no

Exceptions raised at parse time

`SyntaxError`	Something else went wrong during parsing, i.e., during the execution of `readdef`.

Exceptions raised at run time

`NotFound`	A name was looked up in an environment but not found there.
`BindListLength`	A call to `bindList` tried to extend an environment, but it passed two lists (names and values) of different lengths.
`RuntimeError`	Something else went wrong during evaluation, i.e., during the execution of `eval`.

Table 5.1: Exceptions defined especially for this interpreter

5.2 Abstract syntax and values

An abstract-syntax tree can contain a literal value. A value, if it is a closure, can contain an abstract-syntax tree. These two types are therefore mutually recursive, and in ML, two mutually recursive types have to be declared together, using **and**.[1]

215 ⟨*abstract syntax and values* 215⟩≡ (224) 216a▷

```
datatype exp = LITERAL of value
             | VAR     of name
             | SET     of name * exp
             | IFX     of exp * exp * exp
             | WHILEX  of exp * exp
             | BEGIN   of exp list
             | APPLY   of exp * exp list
             | LETX    of let_kind * (name * exp) list * exp
             | LAMBDA  of lambda
and let_kind = LET | LETREC | LETSTAR
and    value = NIL
             | BOOL      of bool
             | NUM       of int
             | SYM       of name
             | PAIR      of value * value
             | CLOSURE   of lambda * value ref env
             | PRIMITIVE of primitive
withtype primitive = value list -> value (* raises RuntimeError *)
     and lambda    = name list * exp

exception RuntimeError of string (* error message *)
```

type env 214
type name 214

These definitions show one refinement of our C implementation; we represent a primitive as an ML function of type `value list -> value`. This simple representation requires no names or tags, and it is suggested by a principle of functional programming: *don't encode a function as data if you can use the function itself.* When we apply a primitive function and something goes wrong, we raise the `RuntimeError` exception.

[1] ML is more restrictive than C, in which we can use incomplete structures to distribute the definitions of mutually recursive types over distant source locations, even in different files. Similar results can be achieved in ML by using "two-level types" (Sheard 2001), but the details are beyond the scope of this book.

For CS45600, Purdue University, Spring 2014 only --- do nc

Definitions are straightforward.

216a ⟨*abstract syntax and values* 215⟩+≡ (224) ◁215

```
datatype def = VAL    of name * exp
             | EXP    of exp
             | DEFINE of name * lambda
             | USE    of name
```

We provide some convenience functions on values. In particular, we show how to embed an ML Boolean or list into its μScheme version and how to render a value as a string.

Embedding and projection An S-expression can represent an integer, Boolean, name, function, list, etc. We may sometimes have an ML Boolean, list, or function that we wish to represent as an S-expression, or similarly, an S-expression that we wish to represent as a value of type `bool`. Here we define mappings between type `value` and some other ML types. Because the set of values representable by an ML value of type `value` strictly contains each of the sets of values representable by these ML types, these mappings are called *embedding* and *projection*. Because the `value` type is strictly larger than these ML types, no embedding operation ever fails, but a projection operation might.[2] For example, although *any* ML function of type `value -> bool` can be embedded into `value` by using the `PRIMITIVE` constructor, there are values of type `value` that cannot be projected into an ML function of type `value -> bool`.

Lists and Booleans are straightforward. Function `embedPredicate` is not a true embedding; it takes any function returning `bool` and returns a corresponding function returning `value`. It really embeds the function's result, not the function itself.

216b ⟨*values* 216b⟩≡ (224) 217a▷

```
embedList      : value list -> value
embedPredicate : ('a -> bool) -> ('a -> value)
bool           : value -> bool
```

```
fun embedList []     = NIL
  | embedList (h::t) = PAIR (h, embedList t)
fun embedPredicate f = fn x => BOOL (f x)
fun bool (BOOL b) = b
  | bool _        = true
```

Function `bool` is the projection function, mapping μScheme values into ML Booleans.

[2] This property is a general characteristic of any embedding/projection pair. Mathematical terminology may clarify; an embedding e of S into S' is an injection from $S \to S'$. The corresponding projection π_e is a left inverse of the embedding; that is $\pi_e \circ e$ is the identity function on S. There is no corresponding guarantee for $e \circ \pi_e$; for example, π_e may be undefined $(\perp)$ on some elements of S', or $e(\pi_e(x))$ may not equal x.

For CS45600, Purdue University, Spring 2014 only --- do no

Printing We render an S-expression as a string.

217a ⟨*values* 216b⟩+≡ (224) ◁216b

```
fun valueString (NIL)    = "()"                        valueString : value -> string
  | valueString (BOOL b) = if b then "#t" else "#f"
  | valueString (NUM n)  = String.map (fn #"~" => #"-" | c => c) (Int.toString n)
  | valueString (SYM v)  = v
  | valueString (PAIR (car, cdr))  =
      let fun tail (PAIR (car, cdr)) = " " ^ valueString car ^ tail cdr
            | tail NIL = ")"
            | tail v = " . " ^ valueString v ^ ")"
      in  "(" ^ valueString car ^ tail cdr
      end
  | valueString (CLOSURE   _) = "<procedure>"
  | valueString (PRIMITIVE _) = "<procedure>"
```

The syntax `Int.toString` indicates the `toString` function from the standard module `Int`. This function, which is part of ML's Standard Basis Library, converts an integer to a string. We use another standard function, `String.map`, to change the minus sign from the ML convention (`~`) to the Scheme convention (`-`).

5.3 Evaluation

The machinery above is enough to write the evaluator, which takes an expression and an environment and produces a value. To make the evaluator easy to write, we do most of the work of evaluation in the nested function `ev`, which inherits the environment `rho` from the outer function `eval`. Because most AST nodes are evaluated in the same environment as their parents, we can evaluate most of them by calling `ev`, which lets `rho` be implicit.

217b ⟨*evaluation* 217b⟩≡ (224) 219b▷

```
⟨definition of separate 218d⟩                 eval : exp * value ref env -> value
fun eval (e, rho) =
  let fun ev (LITERAL n) = n
        ⟨more alternatives for ev 217c⟩
  in  ev e
  end
```

Because `rho` binds names to mutable ref cells, not to values directly, we need the ML functions `!` and `:=` to read and change the contents of the ref cells. The right-hand side of `SET` is evaluated in the same environment as the `SET`, so we use `ev`.

217c ⟨*more alternatives for* ev 217c⟩≡ (217b) 218a▷

```
| ev (VAR v) = !(find(v, rho))
| ev (SET (n, e)) =
    let val v = ev e
    in  find (n, rho) := v;
        v
    end
```


For CS45600, Purdue University, Spring 2014 only --- do nc

Using the projection function bool makes it easy to map μScheme control flow onto ML control-flow constructs.

218a ⟨*more alternatives for* ev 217c⟩+≡ (217b) ◁217c 218b▷

```
  | ev (IFX (e1, e2, e3)) = ev (if bool (ev e1) then e2 else e3)
  | ev (WHILEX (guard, body)) =
      if bool (ev guard) then
        (ev body; ev (WHILEX (guard, body)))
      else
        NIL
  | ev (BEGIN es) =
      let fun b (e::es, lastval) = b (es, ev e)
            | b (   [], lastval) = lastval
      in  b (es, BOOL false)
      end
```

Capturing a closure is as simple as in C. Applying a primitive function is even simpler; we just evaluate the arguments and apply the primitive to the results.

218b ⟨*more alternatives for* ev 217c⟩+≡ (217b) ◁218a 219a▷

```
  | ev (LAMBDA l) = CLOSURE (l, rho)
  | ev (APPLY (f, args))  =
        (case ev f
           of PRIMITIVE prim => prim (map ev args)
            | CLOSURE clo     => ⟨apply closure clo to args 218c⟩
            | v => raise RuntimeError ("Applied non-function " ^ valueString v)
        )
```

Applying a closure is more interesting. To apply a μScheme closure correctly, we have to create fresh locations to hold the values of the actual parameters. In C, we wrote the function allocate for this purpose; in ML, the built-in function ref does the same thing: create a new location and initialize its contents with a value.

⟨*apply closure* clo *to* args 218c⟩≡ (218b)

```
  let val ((formals, body), savedrho) = clo
      val actuals = map ev args
  in  eval (body, bindList (formals, map ref actuals, savedrho))
      handle BindListLength =>
          raise RuntimeError ("Wrong number of arguments to closure; " ^
                              "expected (" ^ spaceSep formals ^ ")")
  end
```

Function separate helps print a readable list of formals for the error message. Function spaceSep is a common special case.

⟨*definition of* separate 218d⟩≡ (217b)

```
  fun separate (zero, sep) =  (* print list with separator *)
    let fun s []     = zero
          | s [x]    = x
          | s (h::t) = h ^ sep ^ s t
    in  s
  end
  val spaceSep = separate ("", " ")  (* print separated by spaces *)
```


For CS45600, Purdue University, Spring 2014 only --- do nc

To interpret `let`, it is easiest to unzip the list of pairs `bs` into a pair of lists `(names, values)`. For `let*`, however, it is easier to walk the bindings one pair at a time. The implementation of `letrec` does both.

The function `ListPair.unzip` is from the `ListPair` module in the Standard Basis Library.

219a ⟨*more alternatives for* `ev` 217c⟩+≡ (217b) ◁218b

```
ListPair.unzip : ('a * 'b) list -> 'a list * 'b list
```

```
| ev (LETX (LET, bs, body)) =
    let val (names, values) = ListPair.unzip bs
    in  eval (body, bindList (names, map (ref o ev) values, rho))
    end
| ev (LETX (LETSTAR, bs, body)) =
    let fun step ((n, e), rho) = bind (n, ref (eval (e, rho)), rho)
    in  eval (body, foldl step rho bs)
    end
| ev (LETX (LETREC, bs, body)) =
    let val (names, values) = ListPair.unzip bs
        val rho' = bindList (names, map (fn _ => ref NIL) values, rho)
        val bs = map (fn (n, e) => (n, eval (e, rho'))) bs
    in  List.app (fn (n, v) => find (n, rho') := v) bs;
        eval (body, rho')
    end
```

5.4 Primitives

Here are the primitives. All are either binary or unary operators. As in C, we would like to reuse the code that does the arity checks. In ML, it's easy; we use higher-order functions to write the arity checks just once. If a check fails, function `arityError` raises `RuntimeError` with a suitable message.

219b ⟨*evaluation* 217b⟩+≡ (224) ◁217b 219c▷

```
unaryOp  : (value         -> value) -> (value list -> value)
binaryOp : (value * value -> value) -> (value list -> value)
```

```
fun arityError n args =
  raise RuntimeError ("primitive function expected " ^ Int.toString n ^
                      "arguments; got " ^ Int.toString (length args))
fun binaryOp f = (fn [a, b] => f (a, b) | args => arityError 2 args)
fun unaryOp  f = (fn [a]    => f a      | args => arityError 1 args)
```

By using higher-order functions, we encapsulate the ideas of "binary operator" and "unary operator" in a general way. As we subdivide our primitives into arithmetic, predicates, list primitives, and other, we use more higher-order functions to specialize things further.

Arithmetic primitives expect and return integers. As in C, we reuse the code that projects two arguments to integers, but as above, we do it using higher-order functions.

219c ⟨*evaluation* 217b⟩+≡ (224) ◁219b 220b▷

```
arithOp: (int * int -> int) -> (value list -> value)
```

```
fun arithOp f = binaryOp (fn (NUM n1, NUM n2) => NUM (f (n1, n2))
                           | _ => raise RuntimeError "integers expected")
```


For CS45600, Purdue University, Spring 2014 only --- do nc

We use the chunk ⟨*primitives* :: 220a⟩ to cons up all the primitives and their names into one giant list. We use that list to build the initial environment for the read-eval-print loop; it plays the same role as the unspeakable `xx` macros in file `prim.h` in the C implementation (chunk 136c).

220a ⟨*primitives* :: 220a⟩≡ (223b) 220c▷
```
("+", arithOp op +  ) ::
("-", arithOp op -  ) ::
("*", arithOp op *  ) ::
("/", arithOp op div) ::
```
The ML keyword `op` makes it possible to use an infix identifier as an ordinary value, so `arithOp op +` passes the value `+` (a binary function) to the function `arithOp`.

We have two kinds of predicate: an ordinary predicate takes one argument and a comparison takes two. Some comparisons apply only to integers. We reuse `embedPredicate` for the definitions.

220b ⟨*evaluation* 217b⟩+≡ (224) ◁219c 221b▷
```
predOp     : (value         -> bool) -> (value list -> value)
comparison : (value * value -> bool) -> (value list -> value)
intcompare : (int   * int   -> bool) -> (value list -> value)
```
```
fun predOp f     = unaryOp  (embedPredicate f)
fun comparison f = binaryOp (embedPredicate f)
fun intcompare f = comparison (fn (NUM n1, NUM n2) => f (n1, n2)
                                | _ => raise RuntimeError "integers expected")
```
Here come the predicates. As required by the semantics of μScheme, equality comparison succeeds only on symbols, numbers, Booleans, and the empty list.

220c ⟨*primitives* :: 220a⟩+≡ (223b) ◁220a 220d▷
```
("<", intcompare op <) ::
(">", intcompare op >) ::
("=", comparison (fn (NIL,     NIL    ) => true
                   | (NUM  n1, NUM  n2) => n1 = n2
                   | (SYM  v1, SYM  v2) => v1 = v2
                   | (BOOL b1, BOOL b2) => b1 = b2
                   | _                  => false)) ::
("null?",    predOp (fn (NIL   ) => true | _ => false)) ::
("boolean?", predOp (fn (BOOL _) => true | _ => false)) ::
("number?",  predOp (fn (NUM  _) => true | _ => false)) ::
("symbol?",  predOp (fn (SYM  _) => true | _ => false)) ::
("pair?",    predOp (fn (PAIR _) => true | _ => false)) ::
("procedure?",
             predOp (fn (PRIMITIVE _) => true | (CLOSURE  _) => true | _ => false)) ::
```
arithOp	219c
binaryOp	219b
BOOL	215
CLOSURE	215
embedPredicate	216b
NIL	215
NUM	215
PAIR	215
PRIMITIVE	215
RuntimeError	215
SYM	215
unaryOp	219b
valueString	217a

The list primitives are easy:

220d ⟨*primitives* :: 220a⟩+≡ (223b) ◁220c 221a▷
```
("cons", binaryOp (fn (a, b) => PAIR (a, b))) ::
("car",  unaryOp  (fn (PAIR (car, _)) => car
                    | v => raise RuntimeError
                                ("car applied to non-list " ^ valueString v))) ::
("cdr",  unaryOp  (fn (PAIR (_, cdr)) => cdr
                    | v => raise RuntimeError
                                ("cdr applied to non-list " ^ valueString v))) ::
```
For CS45600, Purdue University, Spring 2014 only --- do nc

Finally, the only remaining primitives are print and error:

221a ⟨*primitives* :: 220a⟩+≡ (223b) ◁220d

```
("print", unaryOp (fn v => (print (valueString v^"\n"); v)))    ::
("error", unaryOp (fn v => raise RuntimeError (valueString v))) ::
```

5.5 Evaluating definitions

As in Chapter 3, the implementation of the rules for definitions is straightforward. Function `evaldef` takes a definition and an environment, and it returns a new environment. It also takes `echo`, which is not a flag but a function used to print responses. When we see a `val`, we add the name to the environment, evaluate the right-hand side, and print the result of `showVal`, which is either the value of the right-hand side or, in the case of a binding to a function, the name of the function. As usual, `define` is syntactic sugar. When we get an expression, we print its value and bind the result to `it`.

221b ⟨*evaluation* 217b⟩+≡ (224) ◁220b 222b▷

```
evaldef : def  * value ref env * (string->unit) -> value ref env
addName : name * value ref env                  -> value ref env
showVal : name -> exp -> value -> string
```

```
fun evaldef (d, rho, echo) =
  let ⟨definitions of addName and showVal 221c⟩
  in  case d
        of VAL (name, e)         => let val rho = addName (name, rho)
                                        val v   = eval (e, rho)
                                    in  ( find (name, rho) := v
                                        ; echo (showVal name e v)
                                        ; rho
                                        )
                                    end
         | EXP e                 => let val v   = eval (e, rho)
                                        val rho = addName ("it", rho)
                                    in  ( find ("it", rho) := v
                                        ; echo (valueString v)
                                        ; rho
                                        )
                                    end
         | DEFINE (name, lambda) => evaldef (VAL (name, LAMBDA lambda), rho, echo)
         | USE filename          => use readEvalPrint schemeSyntax filename rho
  end
```

bind	214
DEFINE	216a
eval	217b
EXP	216a
find	214
LAMBDA	215
NIL	215
NotFound	214
readEvalPrint	222b
RuntimeError	215
schemeSyntax	674d
unaryOp	219b
USE	216a
use	222a
VAL	216a
valueString	217a

The differences between `VAL` and `EXP` are subtle: for `VAL`, the rules of μScheme require that we add the name to environment `rho` *before* evaluating expression `e`. For `EXP`, we don't bind the name `it` until *after* evaluating the first top-level expression. Also, the results of the two kinds of bindings are printed differently.

The auxiliary functions `addName` and `showVal` are simple:

221c ⟨*definitions of* addName *and* showVal 221c⟩≡ (221b)

```
fun addName (name, rho) = (find (name, rho); rho)
                          handle NotFound _ => bind (name, ref NIL, rho)
fun showVal name (LAMBDA _) _ = name
  | showVal name _          v = valueString v
```

For CS45600, Purdue University, Spring 2014 only --- do nc

The implementation of use is parameterized by readEvalPrint and syntax so we can share it with other interpreters. Function use creates a reader that does not prompt, but it uses writeln to be sure that responses are printed.

222a ⟨*implementation of* use 222a⟩≡ (224)

```
fun use readEvalPrint syntax filename rho =
      let val fd = TextIO.openIn filename
          val defs = reader syntax noPrompts (filename, streamOfLines fd)
          fun writeln s = app print [s, "\n"]
          fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      in  readEvalPrint (defs, writeln, errorln) rho
          before TextIO.closeIn fd
      end
```

Functions reader and streamOfLines are defined in Appendix D. They are based on an abstraction called *streams*. A stream is like a list, except that when client code first looks at an element of a stream, the stream abstraction may do some input or output. Function streamOfLines produces a stream containing the lines of source code found in the named file. Function reader syntax noPrompts converts that stream into a stream of definitions.

5.6 The read-eval-print loop

The read-eval-print loop is built around the inner function processDef, which takes a definition and an environment and produces a new environment. The looping action comes from function streamFold, which applies processDef to every element of a stream of definitions, working from first to last. Function streamFold is the stream analog of the list function foldl.

As in evaldef, echo is not a flag but a function, which takes a string and may print it or do nothing. The argument errmsg is similar to echo, but it is used to issue error messages. Functions evaldef and readEvalPrint are mutually recursive, so we use and instead of fun.

222b ⟨*evaluation* 217b⟩+≡ (224) ◁221b

readEvalPrint : def stream * (string->unit) * (string->unit) -> value ref env -> value ref env

```
and readEvalPrint (defs, echo, errmsg) rho =
  let fun processDef (def, rho) =
            let fun continue msg = (errmsg msg; rho)
            in  evaldef (def, rho, echo)
                handle IO.Io {name, ...} => continue ("I/O error: " ^ name)
                ⟨more read-eval-print handlers 222c⟩
            end
  in  streamFold processDef rho defs
  end
```

evaldef 221b
noPrompts 668d
reader 669
streamFold 650a
streamOfLines 648b

The execution of readdef is protected by a large collection of exception handlers, each of which calls continue to print an error message and to return the (unchanged) environment rho. We reuse the same exception handlers in later interpreters, sometimes with slightly different implementations of continue.

The next handlers deal with problems that arise during I/O, lexical analysis, and parsing.

222c ⟨*more read-eval-print handlers* 222c⟩≡ (222b) 223a▷

The exception IO.Io is part of the Standard Basis Library.

For CS45600, Purdue University, Spring 2014 only --- do no

The remaining handlers deal with problems that arise during evaluation.

223a ⟨*more read-eval-print handlers* 222c⟩+≡ (222b) ◁222c

```
| Div              => continue "Division by zero"
| Overflow         => continue "Arithmetic overflow"
| RuntimeError msg => continue ("run-time error: " ^ msg)
| NotFound n       => continue ("variable " ^ n ^ " not found")
```

Exceptions `Div` and `Overflow` are predefined.

5.7 Initializing and running the interpreter

To make a working interpreter, we need an initial environment. We create the environment by starting with the empty environment, binding the primitive operators, then adding the initial basis.

223b ⟨*initialization* 223b⟩≡ (224) 223c▷

```
fun initialEnv () =
  let val rho =
        foldl (fn ((name, prim), rho) => bind (name, ref (PRIMITIVE prim), rho))
              emptyEnv (⟨primitives :: 220a⟩ nil)
      val basis  = ⟨ML representation of initial basis (automatically generated)⟩
      val defs = reader schemeSyntax noPrompts ("initial basis", streamOfList basis)
  in  readEvalPrint (defs, fn _ => (), fn _ => ()) rho
  end
```

The function `runInterpreter` takes one argument, which tells it whether to prompt. It reads from standard input, again using `streamOfLines`.

223c ⟨*initialization* 223b⟩+≡ (224) ◁223b

`runInterpreter : bool -> unit`

```
fun runInterpreter noisy =
  let val rho = initialEnv ()
      fun writeln s = app print [s, "\n"]
      fun errorln s = TextIO.output (TextIO.stdErr, s ^ "\n")
      val prompts = if noisy then stdPrompts else noPrompts
      val defs =
        reader schemeSyntax prompts ("standard input", streamOfLines TextIO.stdIn)
  in  ignore (readEvalPrint (defs, writeln, errorln) rho)
  end
```

5.8 Building and exporting a program

The final step in implementing the interpreter is to create a function that looks at the command line and calls `runInterpreter`. With a compiler like Moscow ML or MLton, the module isn't *elaborated* until run time, so we can simply insert an irrelevant `val` binding, the elaboration of which has the side effect of calling `main`. `CommandLine.arguments ()` returns an argument list.

223d ⟨*command line* 223d⟩≡ (224)

```
fun main ["-q"] = runInterpreter false
  | main []     = runInterpreter true
  | main _      =
      TextIO.output (TextIO.stdErr, "Usage: " ^ CommandLine.name () ^ " [-q]\n")
val _ = main (CommandLine.arguments ())
```

bind 214
continue,
in Typed Impcore 246b
in Typed μScheme 273b
in μML 330b
in μProlog 557b
in μScheme 222b
in μSmalltalk 494a
emptyEnv 214
noPrompts 668d
NotFound 214
PRIMITIVE 215
reader 669
readEvalPrint 222b
runInterpreter,
in Typed Impcore 247c
in Typed μScheme 274c
in μML 331b
in μSmalltalk 496b
RuntimeError,
in Typed Impcore 244d
in Typed μScheme 268a
in μML 287c
in μProlog 558a
in μScheme 215
in μSmalltalk 469
schemeSyntax 674d
stdPrompts 668d
streamOfLines 648b
streamOfList 647c

For CS45600, Purdue University, Spring 2014 only --- do nc

We build the full interpreter by concatenating the parts in this order:

224 ⟨*mlscheme.sml* 224⟩≡ (0—1)
⟨*environments* 214⟩
⟨*lexical analysis* 671a⟩
⟨*abstract syntax and values* 215⟩
⟨*values* 216b⟩
⟨*parsing* 673a⟩
⟨*implementation of* `use` 222a⟩
⟨*evaluation* 217b⟩
⟨*initialization* 223b⟩
⟨*command line* 223d⟩

For CS45600, Purdue University, Spring 2014 only --- do nc

5.9 Free and bound variables

When an expression e refers to a name y that is introduced outside of e proper, we say that y is *free* in e. We often refer to the set of such names as "free variables," even though "free name" would be more accurate. A variable in e that is introduce within e is a *bound* variable. For example, in the expression

```
(lambda (n) (+ 1 n))
```

the name `+` is a free variable, but `n` is a bound variable. Every variable that appears in an expression is either free or bound.

Each variable that appears in a definition is also free or bound. For example, in

```
(define map (f xs)
  (if (null? xs)
    '()
    (cons (f (car xs)) (map f (cdr xs)))))
```

the names `null?`, `cons`, `car`, and `cdr` are free, and the names `map`, `f`, and `xs` are bound.

Free variables play a key role in implementing closures efficiently. The operational semantics for μScheme say that evaluating a `lambda` expression captures the *entire* environment ρ_c:

$$\frac{x_1, \ldots, x_n \text{ all distinct}}{\langle \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho_c, \sigma \rangle \Downarrow \langle (\!| \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho_c |\!), \sigma \rangle} \quad (\textsc{MkClosure})$$

Do we really need *all* the information in ρ_c? If we look at the application rule to see how ρ_c is used, we can guess maybe not:

$$\frac{\begin{array}{c} \ell_1, \ldots, \ell_n \notin \operatorname{dom} \sigma_n \text{ (and all distinct)} \\ \langle e, \rho, \sigma \rangle \Downarrow \langle (\!| \text{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e_c), \rho_c |\!), \sigma_0 \rangle \\ \langle e_1, \rho, \sigma_0 \rangle \Downarrow \langle v_1, \sigma_1 \rangle \\ \vdots \\ \langle e_n, \rho, \sigma_{n-1} \rangle \Downarrow \langle v_n, \sigma_n \rangle \\ \langle e_c, \rho_c\{x_1 \mapsto \ell_1, \ldots, x_n \mapsto \ell_n\}, \sigma_n\{\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n\} \rangle \Downarrow \langle v, \sigma' \rangle \end{array}}{\langle \text{APPLY}(e, e_1, \ldots, e_n), \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle} \quad (\textsc{ApplyClosure})$$

How is ρ_c used? Only to evaluate the body of the LAMBDA. It turns out that the MKCLOSURE rule need not store all of ρ_c—it is enough to store only those bindings in ρ_c which refer to variables that are free in the LAMBDA expression. Exercise 1 asks you to prove this fact, and Exercise 2 asks you to use it to make the interpreter faster. To do these exercises, you need a precise definition of what a free variable is. We can write such a definition using a proof system.

We'll write a proof system just for expressions; the judgment form is $\boxed{y \in \text{fv}(e)}$. The notation $\text{fv}(e)$ refers to the set of all variables that appear free in e, but we'd rather not construct such a set, so we'll pronounce the judgment $y \in \text{fv}(e)$ as "y appears free in e." To create the proof system, we consider each syntactic form.

A literal expression has no free variables. Formally speaking, no judgment of the form $y \in \text{fv}(\text{LITERAL}v)$ can ever be proved, and we express that fact by not having a rule for literals.

For CS45600, Purdue University, Spring 2014 only — do nc

An expression consisting of a single variable x has just one free variable, which is x itself:

$$\frac{}{x \in \mathrm{fv}(\textsc{var}(x))}$$

The free variables of a SET expression include the free variables being assigned to, plus all the free variables of the right-hand side. So for a SET expression, we have two proof rules:

$$\frac{}{x \in \mathrm{fv}(\textsc{set}(x, e))} \qquad \frac{y \in \mathrm{fv}(e)}{y \in \mathrm{fv}(\textsc{set}(x, e))}$$

A variable is free in an IF expression if and only if it is free in one of the subexpressions:

$$\frac{y \in \mathrm{fv}(e_1)}{y \in \mathrm{fv}(\textsc{if}(e_1, e_2, e_3))} \qquad \frac{y \in \mathrm{fv}(e_2)}{y \in \mathrm{fv}(\textsc{if}(e_1, e_2, e_3))} \qquad \frac{y \in \mathrm{fv}(e_3)}{y \in \mathrm{fv}(\textsc{if}(e_1, e_2, e_3))}$$

A variable is also free in a WHILE expression if and only if it is free in one of the subexpressions:

$$\frac{y \in \mathrm{fv}(e_1)}{y \in \mathrm{fv}(\textsc{while}(e_1, e_2))} \qquad \frac{y \in \mathrm{fv}(e_2)}{y \in \mathrm{fv}(\textsc{while}(e_1, e_2))}$$

And the same for BEGIN:

$$\frac{y \in \mathrm{fv}(e_i)}{y \in \mathrm{fv}(\textsc{begin}(e_1, \ldots, e_n))}$$

A variable is free in an application if and only if it is free in the function or in one of the arguments:

$$\frac{y \in \mathrm{fv}(e)}{y \in \mathrm{fv}(\textsc{apply}(e, e_1, \ldots, e_n))} \qquad \frac{y \in \mathrm{fv}(e_i)}{y \in \mathrm{fv}(\textsc{apply}(e, e_1, \ldots, e_n))}$$

Finally, we get to an interesting case! A variable is free in a LAMBDA expression provided that it is free in the body, and it is *not* one of the arguments:

$$\frac{y \in \mathrm{fv}(e) \qquad y \notin \{x_1, \ldots, x_n\}}{y \in \mathrm{fv}(\textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e))}$$

The rules for the various LET forms require care. The free variables of an ordinary LET are the free variables of the right-hand sides of all the bindings, plus those free variables of the body which are not LET-bound.

$$\frac{y \in \mathrm{fv}(e_i)}{y \in \mathrm{fv}(\textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e))} \qquad \frac{y \in \mathrm{fv}(e) \qquad y \notin \{x_1, \ldots, x_n\}}{y \in \mathrm{fv}(\textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e))}$$

The similarity between the second LET rule and the LAMBDA rule shows a kinship between LET and LAMBDA.

The rule for LETREC is very similar, except that in a LETREC, the bound names x_i are never free:

$$\frac{y \in \mathrm{fv}(e_i) \qquad y \notin \{x_1, \ldots, x_n\}}{y \in \mathrm{fv}(\textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e))} \qquad \frac{y \in \mathrm{fv}(e) \qquad y \notin \{x_1, \ldots, x_n\}}{y \in \mathrm{fv}(\textsc{let}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e))}$$

As is typical, it is a nuisance to try to write the rule for LETSTAR directly. Instead, we treat a LETSTAR expression as a set of nested LET expressions, each containing just one binding. And an empty LETSTAR behaves just like its body.

$$\frac{y \in \mathrm{fv}(\textsc{let}(\langle x_1, e_1\rangle, \textsc{letstar}(\langle x_2, e_2, \ldots, x_n, e_n\rangle, e)))}{y \in \mathrm{fv}(\textsc{letstar}(\langle x_1, e_1, \ldots, x_n, e_n\rangle, e))} \qquad \frac{y \in \mathrm{fv}(e)}{y \in \mathrm{fv}(\textsc{letstar}(\langle\rangle, e))}$$

For CS45600, Purdue University, Spring 2014 only --- do nc

5.10 Exercises

1. In this exercise, you prove that the evaluation of an expression doesn't depend on arbitrary bindings in the environment, but only on the bindings of the expression's free variables.

   If X is a set of variables, we can ask what happens to an environment ρ if we remove the bindings of all the names that are *not* in the set X. The modified environment is written $\rho|_X$, and it is called the *restriction* of ρ to X. The exercise is to prove, by structural induction on derivations, that if $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$, then $\langle e, \rho|_{\mathrm{fv}(e)}, \sigma \rangle \Downarrow \langle v, \sigma \rangle$. To structure the proof, I recommend you introduce a definition and a lemma.

   - Define $\rho \sqsubseteq \rho'$ to mean that $\mathrm{dom}\,\rho \subseteq \mathrm{dom}\,\rho'$, and $\forall x \in \mathrm{dom}\,\rho : \rho(x) = \rho'(x)$. The domain of ρ' contains the domain of ρ, and on their common domain, they agree. We might say that ρ' *extends* ρ.
   - If $X \subseteq X' \subseteq \mathrm{dom}\,\rho$, then $\rho|_X \sqsubseteq \rho|_{X'} \sqsubseteq \rho$.

   These tools are useful, because except for LET forms and LAMBDA expressions, if an expression e has a subexpression e_i, then $\mathrm{fv}(e_i) \subseteq \mathrm{fv}(e)$.

   A reasonable induction hypothesis for the proof might be that if that if $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma \rangle$, and if $\rho|_{\mathrm{fv}(e)} \sqsubseteq \rho'$, then $\langle e, \rho', \sigma \rangle \Downarrow \langle v, \sigma \rangle$.

2. The payoff for the proof in Exercise 1 is that we can use it to optimize code. In chunk 218b, a LAMBDA expression is evaluated by capturing a full environment ρ. Modify the code to capture a restricted environment that contains only the free variables of the LAMBDA expression. That is, instead of allocating the closure $(\!|\mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho|\!)$, allocate $(\!|\mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e), \rho|_X|\!)$, where $X = \mathrm{fv}(\mathrm{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e))$.

3. Change the evaluation of lambda expressions so that the result is a function of type `value list -> value`. On the strength of this trick, rename `PRIMITIVE` to `PROCEDURE`, and eliminate `CLOSURE` from the interpreter.

   What are the drawbacks to this simplification?

4. Revisit the material on proofs and derivations in Section 2.6.

   (a) Devise a representation, in Standard ML, of the judgments of the semantics for μScheme.

   (b) Devise a representation, in Standard ML, of derivations that use the operational semantics of μScheme.

   (c) Change the `eval` function of the μScheme interpreter to return a *derivation* instead of a value.

5. Using the representation of derivations in Exercise 4b, write a *proof checker* that tells whether a given tree represents a valid derivation.

For CS45600, Purdue University, Spring 2014 only --- do nc

For CS45600, Purdue University, Spring 2014 only --- do nc