

μ Scheme in ML

We now turn to a large program written in ML. It is NOT really large, just large enough to implement the μ Scheme interpreter.

It makes use of many of the features of ML, and, since several subsequent exercises will require modifying interpreters written in ML, this seems like the least painful way to begin applying what was, supposedly, learned about ML.

μ Scheme in ML

```
$ mosml -P full
Moscow ML version 2.00 (June 2000)
Enter 'quit();' to quit.
- load "IO";
> val it = () : unit
- use "mlscheme.sml";
[opening file "mlscheme.sml"]
> ...
[closing file "mlscheme.sml"]
> val it = () : unit
-
```

1

2

μ Scheme in ML: Environments

What is a name?

```
- type name = string;
```

```
> type name = string
```

What is an environment?

```
- type 'a env = (name * 'a) list;
```

```
> type 'a env = (string * 'a) list
```

The empty environment:

```
- val emptyEnv = [];
```

```
> val 'a emptyEnv = [] : 'a list
```

```
- val _ = op emptyEnv : 'a env
```

μ Scheme in ML: Environments

Environments as (key, value) pairs:

```
(* lookup and assignment of existing bindings *)
```

```
- exception NotFound of name;
```

```
> exn NotFound = fn : string -> exn
```

```
- fun find (name, []) = raise NotFound name
```

```
  | find (name, (n, v)::tail) = if name = n then v else find(name, tail);
```

```
> val 'a find = fn : string * (string * 'a) list -> 'a
```

```
(* adding new bindings *)
```

```
- exception BindListLength;
```

```
> exn BindListLength = BindListLength : exn
```

```
- fun bind(name, v, rho) = (name, v) :: rho;
```

```
> val ('a, 'b) bind = fn : 'a * 'b * ('a * 'b) list -> ('a * 'b) list
```

3

4

μ Scheme in ML: Environments

Installing a large number of bindings at once.

```
- fun bindList(n::vars, v::vals, rho) = bindList(vars, vals, bind(n, v, rho)
  | bindList([], [], rho) = rho
  | bindList _ = raise BindListLength;
> val ('a, 'b) bindList = fn :
  'a list * 'b list * ('a * 'b) list -> ('a * 'b) list
```

What does this look like?

```
- val testRho = bindList(["a", "b", "c"], [1, 2, 3], []);
> val testRho = [("c", 3), ("b", 2), ("a", 1)] : (string * int) list
- find("a", testRho);
> val it = 1 : int
- find("d", testRho);
! Uncaught exception:
! NotFound "d"
- bindList(["d"], [4, 5], testRho);
! Uncaught exception:
! BindListLength
```

5

μ Scheme in ML: Environments

In theory have **environment** ρ and **store** σ .

Store grows arbitrarily: semantics does not include garbage collection. In implementation: ref provides an immutable pointer to a mutable location. Rather than explicit store have immutable pair of (name, location) where location is mutable.

```
- val x = ref [];
! Warning: Value polymorphism:
! Free type variable(s) at top level in value identifier x
> val x = ref [] : 'a list ref
```

Notice a list ref - but there is a bit of a problem, since the type of the object is too indeterminate. Solution: specify!

```
- val x = ref [] : int list ref;
> val x = ref [] : int list ref
```

6

μ Scheme in ML: Environments

If we want to bind the values 1, 2, 3 to the names “a”, “b” and “c”, we have the function call:

bindList(formals, map ref actuals, savedrho)

At this point, it can be used with an empty initial environment **rho**:

```
- val testRHO = bindList(["a", "b", "c"], map ref [1, 2, 3], []);
> val testRHO = [("c", ref 3), ("b", ref 2), ("a", ref 1)] :
  (string * int ref) list
```

In μ Scheme interpreter, the references will be to an appropriate data structure rather than plain integers.

7

μ Scheme in ML: read-eval-print – Read

1. Lexical analysis: read characters create

```
datatype par = NAME of string
             | INT of int
             | SHARP of char (* #t or #f *)
             | LIST of par list
```

2. Parsing: convert par to abstract syntax.

```
fun parse (NAME s) = VAR s
  | parse (INT n) = LITERAL (NUM n)
  | parse (SHARP c) = LITERAL (BOOL (c = #t))
  | parse (LIST (NAME "if" :: args)) = parseIf args
  | parse (LIST (NAME "begin" :: args)) = BEGIN (map parse args)
...
and parseIf [e1, e2, e3] = IFX (parse e1, parse e2, parse e3)
  | parseIf _ = raise SyntaxError "expected (if e1 e2 e3)"
...
```

8

μ Scheme in ML: Abstract Syntax

```
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
              | LETX     of let_kind * (name * exp) list * exp
              | LAMBDA   of lambda
              | APPLY    of exp * exp list
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
```

9

μ Scheme in ML: read-eval-print – Print

```
fun valueString (NIL) = "()"
  | valueString (BOOL b) = if b then "#t" else "#f"
  | valueString (NUM n) = if n < 0 then "-" ^ Int.toString (~n)
                          else 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>"
(* type declarations for consistency checking *)
val _ = op valueString : value -> string

fun writeln s = app print [s, "\n"]
```

10

μ Scheme in ML: utility routine

```
fun separate (zero, sep) = (* useful for printing *)
  let fun s [] = zero
      | s [x] = x
      | s (h::t) = h ^ sep ^ s t
  in s
  end
```

Example:

```
- separate;
> val it = fn : string * string -> string list -> string
- val separateNums = separate ("EMPTY", ",");
> val separateNums = fn : string list -> string
- "[" ^ separateNums ["1","2","3"] ^ "];
> val it = "[1,2,3]" : string
```

11

μ Scheme in ML – Eval

Over several slides...

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

12

μ Scheme in ML – Eval

```
| ev(BEGIN es) =
  let fun b(e::es, lastval) = b(es, ev e)
      | b([], lastval) = lastval
  in b(es, BOOL false)
  end
| ev(LAMBDA l) = CLOSURE(l, rho)
```

13

μ Scheme in ML – Eval

```
| ev(APPLY (f, args)) =
  (case ev f
   of PRIMITIVE prim => prim (map ev args)
   | CLOSURE clo    =>
      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 (" ^
                separate(" ", " ") formals ^ ")")
          end
      | v => raise RuntimeError ("Applied non-function " ^
                                valueString v)
    )
  )
```

14

μ Scheme in ML

```
| ev(LETX (LET, bs, body)) =
  let val (names, exprs) = ListPair.unzip bs
      val _ = ListPair.unzip : ('a * 'b) list -> 'a list * 'b list
  in eval (body, bindList(names, map (ref o ev) exprs, 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, exprs) = ListPair.unzip bs
      val rho' = bindList(names, map (fn _ => ref NIL) exprs, 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
in ev e
end
val _ = op eval : exp * value ref env -> value
```

15

μ Scheme in ML: more utilities

bool: note different namespaces for types, variables.

```
fun bool (BOOL b) = b
  | bool _ = true
```

Another library routine:

```
- load "ListPair";
> val it = () : unit
- ListPair.unzip [(1, 2), (3, 4), (5, 6)];
> val it = ([1, 3, 5], [2, 4, 6]) : int list * int list
```

16

μ Scheme in ML: primitives

Various arities

```
- 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)

- binaryOp (op +) [1, 2, 3];
! Uncaught exception:
! RuntimeError "primitive function expected 2 arguments; got 3"
```

17

μ Scheme in ML: primitives

Binary arithmetic ops:

```
- fun arithOp f = binaryOp (fn (NUM n1, NUM n2) => NUM (f(n1, n2))
  | _ => raise RuntimeError "integers expected");
> val arithOp = fn : (int * int -> int) -> value list -> value

- arithOp (op +) [NUM 3, NUM 4];
> val it = NUM 7 : value

- arithOp (op +) [SYM "a", NUM 13];
! Uncaught exception:
! RuntimeError "integers expected"
```

18

μ Scheme in ML: primitives

predOp: turn ML predicates into uScheme

```
- fun embedPredicate f = fn x => BOOL (f x)
> val 'a embedPredicate = fn : ('a -> bool) -> 'a -> value
- fun predOp f = unaryOp (embedPredicate f)
> val 'q predOp = fn : ('q -> bool) -> 'q list -> value
- fun positive x = x > 0;
> val positive = fn : int -> bool
- val predPos = predOp positive;
> val predPos = fn : int list -> value
- predPos [1, 2, 3, 4];
! Uncaught exception:
! RuntimeError "primitive function expected 1 arguments; got 4"
- predPos [1];
> val it = BOOL true : value
- predPos [~7];
> val it = BOOL false : value

Similarly binary predicates...
```

19

μ Scheme in ML: primitives

```
fun initialEnv() =
  let val rho =
      foldl (fn ((name, prim), rho) => bind(name, ref (PRIMITIVE prim) , rho))
        emptyEnv
        (("+", arithOp op + ) ::
         ("-", arithOp 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)) ::
         ("cons", binaryOp (fn (a, b) => PAIR(a, b))) ::
         ...
         ("print", unaryOp (fn v => (print (valueString v ^ "\n"); v)))
         ("error", unaryOp (fn v => raise RuntimeError (
           valueString v)))) :: nil)
```

20

μ Scheme in ML: basis library

```
fun initialEnv continued...

val basis = [ "(define caar (lambda (l) (car (car l))))"
, "(define cadr (lambda (l) (car (cdr l))))"
, "(define cdar (lambda (l) (cdr (car l))))"
, "(define list1 (lambda (x) (cons x '())))"
, "(define list2 (lambda (x y) (cons x (list1 y))))"
, "(define list3 (lambda (x y z) (cons x (list2 y z))))"
, "(define length (lambda (l)
, "  (if (null? l) 0
, "      (+ 1 (length (cdr l))))))"
, "(define and (lambda (b c) (if b c b)))"
, "(define not (lambda (b) (if b #f #t)))" ...]

in readEvalPrint
  (topreader (stringsreader basis, false), fn _ => (), fn _ => ()) rho
end
```

21

μ Scheme in ML: loading ρ

Example:

```
readEvalPrint (topreader (stringsreader ["(define fun (x) (+ x x))"], false)
  fn _ => (), fn _ => ()) []; (* note EMPTY ENVIRONMENT *)

> val it =
  [("fun",
    ref(CLOSURE(["x"], APPLY(VAR "+", [VAR "x", VAR "x"])),
      [("fun",
        ref(CLOSURE(["x"], APPLY(VAR "+", [VAR "x", VAR "x"])),
          [("fun",
            ref(CLOSURE(["x"],
              APPLY(VAR "+",
                [VAR "x", VAR "x"])),
              [("fun",
                ref(CLOSURE(#,
                  #)))))))])))] :
  (string * value ref) list

fun is recursive: now have recursive environment.
```

22

μ Scheme in ML: line readers

Return a line or raise EOF

```
type reader = unit -> string (* raises EOF *)
(* readers 573c *)
fun filereader fd () =
  let val line = TextIO.inputLine fd
  in if size line = 0 then raise EOF else line
  end

fun stringsreader l =
  let val buffer = ref l
  in fn () => case !buffer
    of [] => raise EOF
    | h :: t => h before buffer := t
  end

(* type declararations for consistency checking *)
val _ = op filereader : TextIO.instream -> reader
val _ = op stringsreader : string list -> reader
```

23

μ Scheme in ML: readers

top-level reader is a buffer of top-level items, with `nextline` and `firstline` that get more lines and handle prompting:

```
fun readtop (r as { buffer, nextline, firstline }) =
  case !buffer
  of h::t => h before buffer := t
  | [] => ( buffer := map toplevel (read nextline (firstline()))
    ; readtop r
  )
```

Example of before

```
- fun f x = x before print(`Hello World.\n`);
> val 'd f = fn : 'd -> 'd
- f 5;
Hello World
> val it = 5 : int (* returns left argument of before *)
```

24

μ Scheme in ML: readers

```
fun topreader (getline, prompt) =
  let fun promptIn prompt () = ( TextIO.output(TextIO.stdOut, prompt)
                                ; TextIO.flushOut(TextIO.stdOut)
                                ; getline ()
                                )
  in if prompt then
      { buffer = ref [], nextline = promptIn " ",
        firstline = promptIn "-> "
      }
    else
      { buffer = ref [], nextline = getline, firstline = getline }
  end

(* a bit more to handle echoing *)
```

25

μ Scheme in ML: topeval – use

use at top level, reads from file, does not write out prompts.

```
fun use readEvalPrint filename rho =
  let val fd = TextIO.openIn filename
      fun writeln s = app print [s, "\n"]
  in readEvalPrint (topreader (filereader fd, false),
                  writeln, writeln) rho
  before TextIO.closeIn fd
  end
```

26

μ Scheme in ML: topeval

top-level evaluation

```
fun topeval (t, rho, echo) =
  case t
  of USE filename => use readEvalPrint filename rho
  | EXP e          => (echo (valueString (eval(e, rho))); rho)
  | DEFINE (name, lambda) =>
      topeval (VAL (name, LAMBDA lambda), rho, echo)
  | VAL (name, e) =>
      let val rho = (find(name, rho); rho)
          handle NotFound _ => bind (name, ref NIL, rho)
          val v = eval(e, rho)
      in find(name, rho) := v;
        echo (showVal name e v);
        rho
      end
```

27

μ Scheme in ML

Second of three mutually-recursive functions: formatting for echoing.

```
and showVal name (LAMBDA _ ) _ = name
  | showVal name _ v = valueString v
```

28

μ Scheme in ML

Third: loop, handle errors

```
and readEvalPrint (reader, echo, errmsg) rho =
  let fun loop rho =
    let fun continue msg = (errmsg msg; loop rho)
        fun finish () = rho
    in loop (topeval (readtop reader, rho, echo))
      handle EOF => finish()
         | IO.IOException {name, ...} => continue ("I/O error: " ^ name)
         | Paren _ => continue ("syntax error: extra ")
         | PrematureEOF s => (errmsg ("Missing )? Got EOF reading (" ^ s); finish())
         | SyntaxError msg => continue ("syntax error: " ^ msg)
         | Div => continue "Division by zero"
    ...
  end
  in loop rho
  end
```

29

μ Scheme in ML

Put it all together

```
fun runInterpreter noisy =
  let val rho = initialEnv()
      fun writeln s = app print [s, "\n"]
  in ignore (readEvalPrint (topreader (filereader TextIO.stdIn, noisy),
                          writeln, writeln) rho)

      handle EOF => ()
  end
  (* command line *)
fun main ["-q"] = runInterpreter false
  | main [] = runInterpreter true
  | main _ =
    TextIO.output(TextIO.stdErr, "Usage: "
                  ^ CommandLine.name() ^ " [-q]\n")

val _ = main (CommandLine.arguments()) (* run it *)
```

30

μ Scheme in ML

So there it is:

- Minus the lexical analysis.
- A few elipses.
- 20% the size of the C code.

Can you hack it?

31