CS603 Programming Languages 23 February 2005
Department of Computer Science
University of Alabama

The ML code below implements an interpreter for μ-Scheme. Unlike the write-up in chapter five of the textbook, I am including the code that does the parsing. It is interesting use of functional programming. To guide you through the code, I have listed immediately below the important parts of the call graph. For each numbered step, I have placed a marker in the code with that number, along with a hyper link.

①runmain
②runinterpreter
③→filereader: fd -> (unit -> string)
④→topreader: (() -> string, bool) -> topreader
⑤→readEvalPrint: topreader * (string -> unit) * (string -> unit) -> value ref env -> value ref env
⑥→→readtop: topreader -> toplevel
⑦→→→read: (unit -> string) -> string -> par list
⑧→→→→skip: char list -> char list (* removes comments *)
⑨→→→→readline: char list -> par list
⑩→→→→→get: char list -> par * char list

```
type topreader = { buffer: toplevel list ref,
                   nextline: unit -> string,
                   firstline: unit -> string
                 }
```

```
datatype par = NAME of string | INT of int
             | SHARP of char | LIST of par list
```

```
(* mlscheme.sml 195d *)
(* environments 186 *)
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
(* type declararations for consistency checking *)
val _ = op emptyEnv : 'a env
val _ = op find     : name * 'a env -> 'a
val _ = op bind     : name      * 'a      * 'a env -> 'a env
val _ = op bindList : name list * 'a list * 'a env -> 'a env
(* lexical analysis 569 *)
datatype par = NAME of string
             | INT of int
             | SHARP of char  (* #t or #f *)
             | LIST of par list
(* lexical analysis 570a *)

exception Paren of char list  (* raise if found ) when reading input *)
exception PrematureEOF of string (* shows what we were reading ... *)
exception EOF                 (* raised by reader *)
fun read nextline : string -> par list =                              ⑦
  let (* character-fiddling functions 570b *)
      fun isDelim c = List.exists (fn c' => c' = c) [#"(", #")", #" ", #"\t",
                                                     #"\n", #";"]
      (* character-fiddling functions 571a *)
      fun stringReader []     = NONE
        | stringReader (h::t) = SOME (h, t)
      val strton = Int.scan StringCvt.DEC stringReader
      (* type declararations for consistency checking *)
```

CS603 Programming Languages 23 February 2005
Department of Computer Science
University of Alabama

```
val _ = op strton : char list -> (int * char list) option
(* character-fiddling functions 571b *)
fun skip s =
  case StringCvt.skipWS stringReader s
    of #";" :: s => skip (StringCvt.dropl (fn c => c <> #"\n")
                                                    stringReader s)
     | s => s
(* character-fiddling functions 571c *)
fun nonwhite s =
  case skip s
    of [] => nonwhite (explode (nextline()))
     | s  => s
fun readmap f (result, unconsumed) = (f result, unconsumed)
(* get is called only when s is guaranteed non-blank, non-comment, non-
                                                                empty *)
fun get (#"("::s)  = readmap (fn l => LIST l) (list (s, []))
  | get (#")"::s)  = raise (Paren s)
  | get (#"'"::s)  = readmap (fn p => LIST [NAME "quote", p]) (get (
                                                        nonwhite s))
  | get s          = atom (s, [])
and atom(c::s, r) = if isDelim c then (mkAtom(rev r), c::s)
                    else atom(s, c::r)
  | atom([],   r) = (mkAtom(rev r), [])
and mkAtom [#"#", #"f"] = SHARP #"f"
  | mkAtom [#"#", #"t"] = SHARP #"t"
  | mkAtom s = (case strton s of SOME (n, []) => INT n | _ => NAME (
                                                         implode s))
and list (s, l) =
  let val next = nonwhite s
  in  let val (a, u) = get next
      in  list(u, a::l)
      end handle (Paren u) => (rev l, skip u)
               | EOF => raise PrematureEOF (implode next)
  end handle EOF => raise PrematureEOF (implode s)
fun readline [] = []
  | readline s  =
      let val(p, u) = get s
      in  p :: readline(skip u)
      end
(* type declararations for consistency checking *)
val _ = op read : (unit -> string) -> string -> par list
val _ = op get  : char list             -> par      * char list
val _ = op atom : char list * char list -> par      * char list
val _ = op list : char list * par list  -> par list * char list
  in  fn s => readline(skip (explode s))
  end
(* abstract syntax and values 187 *)
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
```

⑧ ⑩ ⑨ ⑦

```
                  | 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 *)
(* abstract syntax and values 188a *)
datatype toplevel = EXP    of exp
                  | DEFINE of name * lambda
                  | VAL    of name * exp
                  | USE    of name
(* values 188b *)



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
(* type declararations for consistency checking *)
val _ = op embedList      : value list -> value
val _ = op embedPredicate : ('a -> bool) -> ('a -> value)
val _ = op bool           : value -> bool
(* values 189a *)
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 declararations for consistency checking *)
val _ = op valueString : value -> string
(* parsing 571d *)
exception SyntaxError of string
fun nodups msg l =
  let fun dup [] = l
        | dup (h::t) = if List.exists (fn x => x = h) t then
                         raise SyntaxError (msg h)
                       else
                         dup t
  in  dup l
  end
fun defineDups f = nodups (fn x => "formal parameter " ^ x ^
                                   " appears twice in definition of function " ^
                                                                          f)
val lambdaDups   = nodups (fn x => "formal parameter " ^ x ^
```

```
                                  " appears twice in lambda")
fun letDups (LETSTAR, bindings) = bindings
  | letDups (kind,    bindings) =
      let val names    = map (fn (n, _) => n) bindings
          val kindName = case kind of LET => "let" | LETREC => "letrec" | _ =>
                                                                        "??"
          fun msg x = "name " ^ x ^ " appears twice in " ^ kindName
      in  nodups msg names;
          bindings
      end
(* parsing 572 *)
fun parse (NAME s)  = VAR s
  | parse (INT n)   = LITERAL (NUM n)
  | parse (SHARP c) = LITERAL (BOOL (c = #"t"))
  | parse (LIST (NAME "quote"  :: args)) = parseQuote args
  | parse (LIST (NAME "if"     :: args)) = parseIf args
  | parse (LIST (NAME "while"  :: args)) = parseWhile  args
  | parse (LIST (NAME "set"    :: args)) = parseSet    args
  | parse (LIST (NAME "begin"  :: args)) = BEGIN (map parse args)
  | parse (LIST (NAME "lambda" :: args)) = parseLambda args
  | parse (LIST (NAME "let"    :: args)) = parseLet LET     args
  | parse (LIST (NAME "letrec" :: args)) = parseLet LETREC  args
  | parse (LIST (NAME "let*"   :: args)) = parseLet LETSTAR args
  | parse (LIST (h::t))  = APPLY (parse h, map parse t)
  | parse (LIST []) = raise SyntaxError "empty application ()"
and parseQuote [e] = LITERAL (quote e)
  | parseQuote _ = raise SyntaxError "expected (quote e)"
and parseIf [e1, e2, e3] = IFX (parse e1, parse e2, parse e3)
  | parseIf _ = raise SyntaxError "expected (if e1 e2 e3)"
and parseWhile [e1, e2] = WHILEX (parse e1, parse e2)
  | parseWhile _ = raise SyntaxError "expected (while e1 e2)"
and parseSet [NAME v, e] = SET (v, parse e)
  | parseSet _ = raise SyntaxError "expected (set v e)"
and parseLambda [LIST formals, body] =
      LAMBDA (lambdaDups (map unname formals), parse body)
  | parseLambda _ =
      raise SyntaxError "expected (lambda (names) body)"
and parseLet kind [LIST bindings, body] =
      LETX(kind, letDups (kind, map unBinding bindings), parse body)
  | parseLet _ _ = raise SyntaxError "expected (let (bindings) body)"
and unname (NAME n) = n
  | unname _          = raise SyntaxError
                              "formal parameters to function must be symbols"
and unBinding (LIST [NAME n, e]) = (n, parse e)
  | unBinding _ = raise SyntaxError "expected (v e) in bindings"
and quote (INT   n) = NUM n
  | quote (SHARP c) = BOOL (c = #"t")
  | quote (NAME  ".") =
      raise SyntaxError
                  "this interpreter cannot handle . in quoted S-expressions"
  | quote (NAME  n) = SYM n
  | quote (LIST l) = embedList (map quote l)
(* parsing 573a *)
fun toplevel (LIST (NAME "define" :: args)) = parseDefine args
  | toplevel (LIST (NAME "val" :: args))    = parseVal args
  | toplevel (LIST (NAME "global" :: args)) = parseVal args (* legacy *)
  | toplevel (LIST (NAME "use"    :: args)) = parseUse    args
  | toplevel e = EXP (parse e)
```

```
and parseDefine [NAME name, LIST formals, body] =
      DEFINE (name, (defineDups name (map unname formals), parse body))
  | parseDefine _ = raise SyntaxError "expected (define f (args) body)"
and parseVal [NAME v, e] = VAL (v, parse e)
  | parseVal _ = raise SyntaxError "expected (val v e)"
and parseUse [NAME v] = USE v
  | parseUse _ = raise SyntaxError "expected (use filename)"
(* readers 573b *)
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
(* readers 574a *)
type topreader = { buffer    : toplevel list ref
                 , nextline  : unit -> string
                 , firstline : unit -> string
                 }
fun readtop (r as { buffer, nextline, firstline }) =
  case !buffer
    of h::t => h before buffer := t
     | []   => ( buffer := map toplevel (read nextline (firstline()))
               ; readtop r
               )
(* readers 574b *)
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
(* readers 574c *)
fun echoTag f x =
  let val line = f x
      val _ = if (String.substring (line, 0, 2) = ";#" handle _ => false) then
                print line
              else
                ()
  in  line
  end

fun echoBuf { buffer=b, nextline=n, firstline=f } =
```

CS603 Programming Languages 23 February 2005
Department of Computer Science
University of Alabama

```
 { buffer=b, nextline=echoTag n, firstline=echoTag f }

val topreader = fn args => echoBuf (topreader args)
(* implementation of [[use]] 193c *)
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
(* evaluation 189b *)
(* definition of [[separate]] 190d *)
fun separate (zero, sep) =  (* useful for printing *)
  let fun s []     = zero
        | s [x]    = x
        | s (h::t) = h ^ sep ^ s t
  in  s
end
fun eval(e, rho) =
  let fun ev(LITERAL n) = n
        (* more alternatives for [[ev]] 189c *)
        | ev(VAR v) = !(find(v, rho))
        | ev(SET (n, e)) =
            let val v = ev e
            in  find (n, rho) := v;
                v
            end
        (* more alternatives for [[ev]] 190a *)
        | 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
        (* more alternatives for [[ev]] 190b *)
        | 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]] 190c *)
                                   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)
              )
```

```
        (* more alternatives for [[ev]] 191a *)
        | ev(LETX (LET, bs, body)) =
            let val (names, values) = ListPair.unzip bs
        (* type declararations for consistency checking *)
        val _ = ListPair.unzip : ('a * 'b) list -> 'a list * 'b list
            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
  in  ev e
  end
(* type declararations for consistency checking *)
val _ = op eval : exp * value ref env -> value
(* evaluation 191b *)
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)
(* type declararations for consistency checking *)
val _ = op unaryOp  : (value         -> value) -> (value list -> value)
val _ = op binaryOp : (value * value -> value) -> (value list -> value)
(* evaluation 191c *)
fun arithOp f = binaryOp (fn (NUM n1, NUM n2) => NUM (f(n1, n2))
                           | _ => raise RuntimeError "integers expected")
(* type declararations for consistency checking *)
val _ = op arithOp: (int * int -> int) -> (value list -> value)
(* evaluation 192b *)
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")
(* type declararations for consistency checking *)
val _ = op predOp     : (value         -> bool) -> (value list -> value)
val _ = op comparison : (value * value -> bool) -> (value list -> value)
val _ = op intcompare : (int   * int   -> bool) -> (value list -> value)
(* evaluation 193b *)
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
and showVal name (LAMBDA _) _ = name
  | showVal name _          v = valueString v
(* evaluation 194a *)
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()
                   (* more read-eval-print handlers 194b *)
                   | IO.Io {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)
                   (* more read-eval-print handlers 194c *)
                   | Div               => continue "Division by zero"
                   | Overflow          => continue "Arithmetic overflow"
                   | RuntimeError msg  => continue ("run-time error: " ^ msg)
                   | NotFound n        => continue ("variable " ^ n ^
                                                          " not found")
        end
  in  loop rho
  end
(* type declararations for consistency checking *)
val _ = op topeval : toplevel * value ref env * (string->unit) -> value ref env
(* type declararations for consistency checking *)
val _ = op readEvalPrint : topreader * (string->unit) * (string->unit) -> value
                                                    ref env -> value ref env
(* initialization 195a *)
fun initialEnv() =
  let val rho = foldl (fn ((name, prim), rho) => bind(name, ref (PRIMITIVE prim)
                                                                  , rho))
                      emptyEnv ((* primitives [[::]] 192a *)
                                ("+", arithOp op +  ) ::
                                ("-", arithOp op -  ) ::
                                ("*", arithOp op *  ) ::
                                ("/", arithOp op div) ::
                                (* primitives [[::]] 192c *)
                                ("<", 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)) ::
```

⑤

```
                              ("symbol?",  predOp (fn (SYM  _) => true | _ =>
                                                                    false)) ::
                              ("pair?",    predOp (fn (PAIR _) => true | _ =>
                                                                    false)) ::
                              ("procedure?",
                                          predOp (fn (PRIMITIVE _) => true |
                                        (CLOSURE  _) => true | _ => false)) ::
                              (* primitives [[::]] 192d *)
                              ("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))) ::
                              (* primitives [[::]] 193a *)
                              ("print", unaryOp (fn v => (print (valueString v
                                                                  ^"\n"); v))) ::
                              ("error", unaryOp (fn v => raise RuntimeError (
                                                       valueString v))) :: nil)
  in  readEvalPrint
      (topreader (stringsreader (* ML representation of initial basis *)
                                 [ "(define caar (l) (car (car l)))"
                                 , "(define cadr (l) (car (cdr l)))"
                                 , "(define cdar (l) (cdr (car l)))"
                                 , "(define list1 (x)     (cons x '()))"
                                 , "(define list2 (x y)   (cons x (list1 y)))"
                                 ,
                                  "(define list3 (x y z) (cons x (list2 y z)))"
                                 , "(define length (l)"
                                 , "  (if (null? l) 0"
                                 , "    (+ 1 (length (cdr l)))))"
                                 , "(define and (b c) (if b  c  b))"
                                 , "(define or  (b c) (if b  b  c))"
                                 , "(define not (b)   (if b #f #t))"
                                 ,
"(define atom? (x) (or (number? x) (or (symbol? x) (or (boolean? x) (null? x)))))"
                                 , "(define equal? (s1 s2)"
                                 , "  (if (or (atom? s1) (atom? s2))"
                                 , "    (= s1 s2)"
                                 ,
              "    (and (equal? (car s1) (car s2)) (equal? (cdr s1) (cdr s2)))))"
                                 , "(define append (l1 l2)"
                                 , "  (if (null? l1)"
                                 , "     l2"
                                 ,
                                  "     (cons (car l1) (append (cdr l1) l2))))"
                                 , "(define revapp (l1 l2)"
                                 , "  (if (null? l1)"
                                 , "     l2"
                                 ,
                                  "     (revapp (cdr l1) (cons (car l1) l2))))"
                                 , "(define bind (x y alist)"
                                 , "  (if (null? alist)"
                                 , "    (list1 (list2 x y))"
                                 , "    (if (equal? x (caar alist))"
```

```
           , "      (cons (list2 x y) (cdr alist))"
           ,
    "      (cons (car alist) (bind x y (cdr alist))))))"
           , "(define find (x alist)"
           , "  (if (null? alist) '()"
           , "    (if (equal? x (caar alist))"
           , "      (car (cdar alist))"
           , "      (find x (cdr alist)))))"
           , "(define o (f g) (lambda (x) (f (g x))))"
           ,
"(define curry   (f) (lambda (x) (lambda (y) (f x y))))"
           ,
          "(define uncurry (f) (lambda (x y) ((f x) y)))"
           , "(define filter (p? l)"
           , "  (if (null? l)"
           , "    '()"
           , "    (if (p? (car l))"
           , "      (cons (car l) (filter p? (cdr l)))"
           , "      (filter p? (cdr l)))))"
           , "(define map (f l)"
           , "  (if (null? l)"
           , "    '()"
           , "    (cons (f (car l)) (map f (cdr l)))))"
           , "(define exists? (p? l)"
           , "  (if (null? l)"
           , "    #f"
           , "    (if (p? (car l)) "
           , "      #t"
           , "      (exists? p? (cdr l)))))"
           , "(define all? (p? l)"
           , "  (if (null? l)"
           , "    #t"
           , "    (if (p? (car l))"
           , "      (all? p? (cdr l))"
           , "      #f)))"
           , "(define foldr (op zero l)"
           , "  (if (null? l)"
           , "    zero"
           , "    (op (car l) (foldr op zero (cdr l)))))"
           , "(define foldl (op zero l)"
           , "  (if (null? l)"
           , "    zero"
           , "    (foldl op (op (car l) zero) (cdr l))))"
           , "(define <= (x y) (not (> x y)))"
           , "(define >= (x y) (not (< x y)))"
           , "(define != (x y) (not (= x y)))"
           , "(define max (x y) (if (> x y) x y))"
           , "(define min (x y) (if (< x y) x y))"
           , "(define mod (m n) (- m (* n (/ m n))))"
           ,
  "(define gcd (m n) (if (= n 0) m (gcd n (mod m n))))"
           , "(define lcm (m n) (* m (/ n (gcd m n))))"
           , "(define caar (x) (car (car x)))"
           , "(define cdar (x) (cdr (car x)))"
           , "(define cadr (x) (car (cdr x)))"
           , "(define cddr (x) (cdr (cdr x)))"
           , "(define caaar (x) (car (caar x)))"
           , "(define cdaar (x) (cdr (caar x)))"
```

```
                                , "(define caadr (x) (car (cadr x)))"
                                , "(define cdadr (x) (cdr (cadr x)))"
                                , "(define cadar (x) (car (cdar x)))"
                                , "(define cddar (x) (cdr (cdar x)))"
                                , "(define caddr (x) (car (cddr x)))"
                                , "(define cdddr (x) (cdr (cddr x)))"
                                ,
                               "(define list1 (x)              (cons x '()))"
                                ,
                             "(define list2 (x y)            (cons x (list1 y)))"
                                ,
                           "(define list3 (x y z)          (cons x (list2 y z)))"
                                ,
                         "(define list4 (x y z a)        (cons x (list3 y z a)))"
                                ,
                       "(define list5 (x y z a b)      (cons x (list4 y z a b)))"
                                ,
                     "(define list6 (x y z a b c)    (cons x (list5 y z a b c)))"
                                ,
                   "(define list7 (x y z a b c d)  (cons x (list6 y z a b c d)))"
                                ,
                 "(define list8 (x y z a b c d e) (cons x (list7 y z a b c d e)))"
                                ], false),
       fn _ => (), fn _ => ()) rho
  end
(* initialization 195b *)
fun runInterpreter noisy =
  let val rho = initialEnv()                                          ②
      fun writeln s = app print [s, "\n"]
  in  ignore (readEvalPrint (topreader (filereader TextIO.stdIn, noisy),
                             writeln, writeln) rho)
      handle EOF => ()
  end
(* type declararations for consistency checking *)
val _ = op runInterpreter : bool -> unit
(* command line *)
fun runmain ["-q"] = runInterpreter false
  | runmain []     = runInterpreter true                            ①
  | runmain _      =
      TextIO.output(TextIO.stdErr, "Usage: " ^ CommandLine.name() ^ " [-q]\n")
val _ = runmain (CommandLine.arguments())
```