```
Types

datatype top_level =
                 Exp_t    of exp
               | Decl_t   of decl list
(* Declarations *)
datatype decl = Val_d     of (id * typ option * exp) (* val x = (5,2) *)
              | Fun_d     of (funrec * exp)     (* fun inc(i:int):int = i+1 *)

type funrec = {name:id, args: (id * typ option) list, ret_typ: typ}

datatype typ = Int_t
             | Bool_t
             | Unit_t
             | Tuple_t   of typ list
             | List_t    of typ
             | Fn_t      of (typ * typ)

(* Expressions *)
datatype exp = Int_c     of int                (* 17      *)
             | Real_c    of real               (* 12.73   *)
             | Bool_c    of bool               (* true    *)
             | Char_c    of char               (* #"a"    *)
             | String_c  of string             (* "alpha" *)
             | Unit_c                          (* ()      *)
             | Id_e      of id                 (* any variable identifier  *)
             | If_e      of (exp * exp * exp)  (* if b then a else b       *)
             | Let_e     of (decl list * exp)  (* let val x = 4 in x+x end *)
             | Fn_e      of ((id*typ option)list * typ option * exp)
                                               (* fn (s:int):int=>6 *)
             | Apply_e   of (exp * exp)        (* increment(6)      *)
             | Unop_e    of (unop * exp)       (* not true *)
             | Binop_e   of (exp * binop * exp) (* 5 + 5    *)
             | Tuple_e   of (exp list)         (* (5,4,4,3)  *)
             | Ith_e     of (int * exp)        (* #1 (3,4,5) *)
             | List_e    of (exp list)         (* [1, 2] *)

(* Values *)
datatype
    value =
    Int_v      of int
  | Bool_v     of bool
  | Unit_v
  | Tuple_v    of value list
  | List_v     of value list
  | Fn_v       of (string option     (* function name        *)
                 * exn               (* environment          *)
                 * string list       (* names of arguments   *)
                 * exp)              (* body                 *)
  | Predef_v   of string      (* name of predefined function *)
  | SpecForm_v   of string      (* name of predefined function *)
  | Thunk_v    of exp * env         (* for lazy evaluation  *)

and env = Env of (string * value) list

Structures

structure AbstractSyntax = struct
  type     id    = string
  datatype typ   = ...
  datatype binop = ...
  datatype unop  = ...
  datatype exp   = ...
       and decl  = ...
  datatype top_level = ...

  exception TypeUnification

  fun unifyTypes (t: typ, t': typ): typ = ...
end

structure Interpreter =
struct
  fun evalStr(env: Environment.env, str: string): Environment.env = let
      val newEnv = (case Parser.parseString str of
                      NONE ⇒ env
                    | SOME (Exp_t exp) ⇒
                        (Printer.show
                         (forceValue(Evaluator.evaluate(exp,env))));
                        env)
                    | SOME (Decl_t []) ⇒ env
                    | SOME (Decl_t dlist) ⇒ (foldl Evaluator.evaluateDeclare
                                                   env dlist))
    in
      newEnv
    end
  ...
  fun mainLoop(env: Environment.env): unit =
    let
      val () = print "Mini-ML => "
      val line = TextIO.inputLine TextIO.stdIn
    in
      if (size line) >= 2 andalso String.sub(line,0) = #":" then
        case String.sub(line, 1) of
          #"h" ⇒ (showHelp(); mainLoop(env))
          ...
      else mainLoop(evalStr(env, line))
    end
  ...
  fun run(): unit = let
    val () = mainLoop (Environment.topLevel)
    ...
  in
    ()
  end

end

structure Environment = struct
  datatype value = ...
       and env   = Env of (string * value) list
  val topLevel = Env([
    ("hd",   Predef_v ("hd")),
    ("null", Predef_v ("null")),
    ...
    ("if3",  SpecForm_v ("if3"))
  ])

  fun lookupBinding (Env(e)) (id:string): value option = ...
  fun insertBinding (Env(e)) (id:string) (v: value): env = ...
end

structure Evaluator = ...
  exception ClosureEnv of env

  fun evaluate (exp : exp, env : env) =
    case exp of
      Int_c i ⇒ Int_v i
    ...
    | Id_e i ⇒ (case (lookupBinding env i) of
                  SOME v ⇒ v
                | NONE ⇒ Error.runtime ("binding " ^ i ^ " not found"))
    | If_e (test, e1, e2) ⇒ (case forceValue(evaluate(test, env)) of
                               Bool_v true ⇒ evaluate(e1, env)
                             | Bool_v false ⇒ evaluate(e2, env)
                             | _ ⇒ Error.runtime
                                "condition of if not a boolean")
    | Let_e (decls, e) ⇒ let
        val newEnv = foldl evaluateDeclare env decls
      in
        evaluate(e, newEnv)
      end
    | Fn_e (args, ret, e) ⇒ let
        val (names, types) = ListPair.unzip args
      in
        Fn_v(NONE, ClosureEnv env, names, e)
      end
    | Apply_e (func, arg) ⇒ evaluateApply(func, arg, env)
    | Unop_e(unop, e) ⇒ doUnop(unop, e, env)
    | Binop_e(e1, binop, e2) ⇒ doBinop(e1, binop, e2, env)
    | Tuple_e tl ⇒ Tuple_v (map (fn e ⇒ evaluate(e,env)) tl)
    | Ith_e (ind, tuple) ⇒ (case evaluate(tuple,env) of
       Tuple_v l ⇒ (List.nth(l,ind - 1) (* stupid 1-based indexing *)
         handle Subscript ⇒ Error.runtime "tuple projection out of bounds")
      | _ ⇒ Error.runtime "projection applied to non-tuple")
    | List_e el ⇒ List_v (map (fn e ⇒ evaluate(e,env)) el)

  and doUnop(unop: unop, e: exp, env: env): value =
```

```
    let
      val arg = evaluate(e, env)
    in
      case (unop, arg) of
        (Neg, Int_v i) ⇒ Int_v(~i)
      | (Not, Bool_v b) ⇒ Bool_v(not b)
      | _ ⇒ Error.runtime "error in applying unary operator"
    end

  and doBinop(e1: exp, binop: binop, e2: exp, env: env): value =
    case binop of
      (* These primitives do not force their arguments *)
      AndAlso ⇒ (case forceValue(evaluate(e1, env)) of
                   Bool_v false ⇒ Bool_v false
                 | Bool_v true ⇒ let
                     val result = forceValue(evaluate(e2, env))
                   in
                     case result of
                       Bool_v _ ⇒ result
                     | _ ⇒ Error.runtime
                         "second part of andalso has wrong type"
                   end
                 | _ ⇒ Error.runtime "first part of andalso has wrong type")
    | OrElse ⇒ (case forceValue(evaluate(e1, env)) of ...
    (* All other primitives force their arguments *)
    | _ ⇒ let
        val v1 = forceValue(evaluate(e1, env))
        val v2 = forceValue(evaluate(e2, env))
      in
        case (v1, binop, v2) of
          (Int_v i1, Plus, Int_v i2) ⇒ Int_v (i1 + i2)
        | (Int_v i1, Minus, Int_v i2) ⇒ Int_v (i1 - i2)
        ...
      end
  and evaluateDeclare (decl: AbstractSyntax.decl, env: Environment.env): env =
    case decl of
      Val_d(name, t, exp) ⇒ insertBinding env name (evaluate (exp,env))
    | Fun_d _  ⇒ Error.runtime "fun not supported"

  and evaluateApply(func: exp, arg: exp, env: env) =
    let
      val f = forceValue(evaluate(func, env))
      val normalorder = false         (* Normal versus applicative order *)
      fun evalAsUsual() =
        let
          fun process(ex:exp) =
            (* In applicative order we never introduce thunks *)
            if normalorder then Thunk_v(ex,env) else evaluate(ex,env)
          val arglist =
            case arg of
                  Tuple_e alist ⇒ map process alist
                | Unit_c ⇒ [Unit_v]
                | _ ⇒ [process arg]
        in
          case f of
            Fn_v(_) ⇒ apply(f,arglist)
          | Predef_v (name) ⇒ applyPredef(name,arglist)
          | _ ⇒ Error.runtime "attempted to apply non-function"
        end
      fun evalSpecial(name: string) =
        case (name,arg) of
          ("if3", Tuple_e [arg1, arg2, arg3]) ⇒
            (case evaluate(arg1,env) of
               Bool_v true ⇒ evaluate(arg2, env)
             | Bool_v false ⇒ evaluate(arg3, env)
             | _ ⇒ Error.runtime "test must be boolean")
        | _ ⇒ Error.runtime "no handler for this special form"
    in
      case f of
        SpecForm_v(name) ⇒ evalSpecial(name)
        | _ ⇒ evalAsUsual()
    end

  (* Computes a value from expressions that might contain thunks. *)
  and forceValue (v: value) =
    case v of
      Thunk_v(exp,env) ⇒
        let
          val ans = forceValue(evaluate(exp,env))
        in
          ans
        end
    | _ ⇒ v

  and apply (func: value, args: value list): value =
    case func of
      Fn_v(NONE, ClosureEnv env, names, exp) ⇒ let
        val () = if (length names) = (length args) then ()
                 else Error.runtime "bad arguments to anonymous function"
        val fullEnv = foldl (fn ((id,v),e) ⇒ insertBinding e id v)
                            env (ListPair.zip (names, args))
      in
        evaluate(exp, fullEnv)
      end
    | Fn_v(SOME name, ClosureEnv env, names, exp) ⇒
        Error.runtime "named functions not supported"
    | Predef_v(name) ⇒ applyPredef(name, args)
    | _ ⇒ Error.runtime "apply attempted with non-function"

  and applyPredef(name: string, args: value list): value =
     let val forcedargs = map forceValue args
     in
       case (name,forcedargs) of
         ("hd", [List_v l]) ⇒ (hd l handle Empty ⇒
                                 Error.runtime "tried to take hd of []")
       | ("tl", [List_v l]) ⇒ (List_v (tl l) handle Empty ⇒
                                 Error.runtime "tried to take tl of []")
       | ("null", [List_v l]) ⇒ Bool_v(List.null l)
       | _ ⇒ Error.runtime
           ("unknown function or incorrect parameters: " ^ name)
     end


end
```