The code below contains the essential parts of the functional MiniML interpreter. It has been simplified slightly so that the eager evaluation and scoping rules are now hardcoded. The code made redundant by this change has been eliminated (e.g. thunks, function `forceValue`, variables `scoping` and `evaluation`).

```plaintext
(* The code below contains the essential parts of the functional MiniML evaluator. It has been simplified slightly so that the eager evaluation mode and scoping rules are now hardcoded. The code made redundant by this change has been eliminated (e.g. thunks, function `forceValue`, variables `scoping` and `evaluation`). *)

structure Interpreter =

  struct
  fun loop (en, env, prenv: bool):unit = ...

  let
    val t = (parseString:string→AbstractSyntax.top_level option)inline
    in
      case t of
        SOME(Exp_t ex) ⇒ (print(printValue(evaluate(ex, en), 0));
          loop(en, prenv))
        SOME(Decl_t dlist) ⇒ let
          val env = evaluateDeclare(dlist, en)
          in
            if print then print(printEnv(en, 0))
            else ()
            end
        end
        handle Error.Error ⇒ loop(en, prenv)
      end

  end

structure Environment = struct
  val top_level = Env([...
  end

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 Environment = struct
  val top_level = Env([...
  end

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
```

---

```
(* Structures *)

structure Interpreter =

  struct
  fun loop (en, env, prenv: bool):unit = ...

  let
    val t = (parseString:string→AbstractSyntax.top_level option)inline
    in
      case t of
        SOME(Exp_t ex) ⇒ (print(printValue(evaluate(ex, en), 0));
          loop(en, prenv))
        SOME(Decl_t dlist) ⇒ let
          val env = evaluateDeclare(dlist, en)
          in
            if print then print(printEnv(en, 0))
            else ()
            end
        end
        handle Error.Error ⇒ loop(en, prenv)
      end

  end

structure Environment = struct
  val top_level = Env([...
  end

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
```
evaluate ('then' branch) ⇒
| evaluate (thenE, en)
| (Bool_v true, Bool_t) ⇒ (* evaluate 'then' branch *)
evaluate (thenE, en)
| (Bool_v false, Bool_t) ⇒ (* evaluate 'else' branch *)
evaluate (elseE, en)
| _ ⇒ err "first argument of if must be boolean"

⇒
(* evaluate 'else' branch *)

u = unified
evaluate (elseE, en)

| _ ⇒ err "incorrect argument number for if; should be 3"

...⇒

and evaluate (ex: exp, en: env): value * typ =

...⇒

case ex of
Int_v l ⇒ (Int_v l, Int_t)
| _ ⇒ err "argument types don't match in function call"

let
(* Eager evaluation! Evaluate function, check returned type (r = returned) *)
val (rv, rt) = evaluate (e1, encrt)
val (fc, fct) = evaluate (e1, encrt)
in
val urt = unifyTypes (fct, rt)
handle TypeUnification ⇒
err "argument types don't match in function call"
in
(* Are there too few arguments? *)
if List.length fal2 > 0
then (* this is a curried function ⇒ return closure *)
  Fn_v (fatl2, ListPair.foldl (fn, encrt)
  InsertBinding (fa, (a, at), en')
  en
  (ListPair.zip (fall, at), utl),
  body,
  NONE),
  Fn_t (Tuple_t fatl2, frt))
else (*
  Eager evaluation! Evaluate function, check returned type (r = returned) *)
  let
    val (rv, rt) = evaluate (body,
      ListPair.foldl (fn, encrt)
      InsertBinding (fa, (a, at), en')
      en
      (ListPair.zip (fall, at), utl))
    val urt = unifyTypes (fct, rt)
    handle TypeUnification ⇒
    err "argument types don't match in function call"
in
(* We are using static scoping! *)
val (rv, rt) = evaluate (body,
  ListPair.foldl (fn, encrt)
  InsertBinding (fa, (a, at), en')
  en
  (ListPair.zip (fall, at), utl))
val (fc, fct) = evaluate (e1, encrt)
in
(* Evaluate = eagerly! = the function’s arguments *)
val (a, at) = evaluate (e2, encrt)
(* Transfer values & types of actual parameters into lists. *)
val (al, atl) = case (a, at) of
  (Tuple_v al, Tuple_t atl) ⇒ (al, atl)
| _ ⇒ err "Too many arguments provided in function call"
(* First, retrieve the types of the formal arguments. *)
val (fatl, frt) = case fct of
  Fn_t (Tuple_t fatl, frt) ⇒ (fatl, frt)
| _ ⇒ err "incorrect number of arguments"

(* Are there too many arguments? That would be bad... *)
val _ = if List.length al2 > 0
  then err "too many arguments provided in function call"
  else ()