Types

```
top_level = Exp_t  of exp
          | Decl_t of decl list

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

(* Declarations *)
decl = Val_d    of (id * typ * exp)
     | Fun_d    of (funrec * exp)

(* Types *)
typ = Int_t
    | Real_t
    | Bool_t
    | Char_t
    | String_t
    | Tuple_t   of typ list
    | List_t    of typ
    | Fn_t      of (typ * typ)
    | Ref_t     of typ
    | Undef_t

(* Expressions *)
exp = Int_c     of int
    | Real_c    of real
    | Bool_c    of bool
    | Char_c    of char
    | String_c  of string
    | Id_e      of id
    | If_e      of (exp * exp * exp)
    | Let_e     of (decl list * exp)
    | Fn_e      of ((id*typ)list * typ * exp)
    | Apply_e   of (exp * exp)
    | Unop_e    of (unop * exp)
    | Binop_e   of (exp * binop * exp)
    | Tuple_e   of (exp list)
    | Ith_e     of (int * exp)
    | List_e    of (exp list)

(* Values *)
value = Int_v   of int
    | Real_v    of real
    | Bool_v    of bool
    | Char_v    of char
    | String_v  of string
    | Tuple_v   of value list
    | List_v    of value list
    | Fn_v      of (string list * env * exp * string option)
    | Predef_v   of string
    | SpecForm_v of string
    | Thunk_v    of exp * env
    | Dyn_v      of exp

and env = Env of (string * value * typ) 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 loop (en: env, prenv: bool):unit =
  ...
        let
          val t = (parseString:string→AbstractSyntax.top_level option)inLine
        in
          (case t of
             NONE              ⇒ loop(en, prenv)
           | SOME(Exp_t ex)    ⇒ (print(printValue(
                                        forceValue(evaluate (ex, en)),
                                        0));
                                  loop(en, prenv))
           | SOME(Decl_t dlist) ⇒ let
                                     val en = evaluateDeclare(dlist, en)
                                   in
                                     (if prenv then print(printEnv(en, 0))
                                              else ();
                                      loop(en, prenv))
                                   end)
             handle Error.Error ⇒ loop(en, prenv)
        end
    ...
end

structure Environment = struct
  datatype value = ...
       and env   = Env of (string * value * typ) list

  val top_level = Env([
    ("stmtlst",   Predef_v("stmtlst"),     Fn_t(Undef_t, Undef_t)),
    ("ncat",      Predef_v("ncat"),        Fn_t(Undef_t, String_t)),
    ...
  ])

  fun lookupBinding (id: string, Env(en)): (value * typ) option = ...
  fun insertBinding (id: string, (v, t): value * typ, Env(en)): env = ...
end

structure Evaluator =

  (* Variables that control the evaluator. *)
  val debug:  bool = false
  datatype scoping_style = STATIC | DYNAMIC
  val scoping: scoping_style = STATIC
  datatype evaluation_style = LAZY | EAGER
  val evaluation: evaluation_style = LAZY

  fun predefined (name: string, (arg, argt): value * typ): value * typ =
  (
    ...
    | ("ncat",  Tuple_v sl, Tuple_t tl  )  ⇒
        if List.all (fn t ⇒ case t of String_t ⇒ true | _ ⇒ false) tl
        then
          (String_v (foldl (fn (sv, cs) ⇒ case sv of
                                             String_v s ⇒ cs ^ s
                                           | _ ⇒ err "internal error [10]")
                           ""
                           sl),
           String_t)
        else
          err "'ncat' takes only strings"
    (* If a function has a single argument, that is transmitted as such,
       not as a tuple. Zero or more than one argument results in tuples. *)
    | ("ncat", String_v _, String_t     )   ⇒ (arg, argt)
    | ("ncat", _,          _            )   ⇒ err "ncat needs strings"
    ...
  )

  fun specialForm (name: string, expr: exp, en: env): value * typ =
  (
    ...
    case name of
      "if3" ⇒
         (case expr of
            Tuple_e([cond, thenE, elseE]) ⇒
             (case evaluate(cond, en) of
                (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 if3 must be boolean")
          | _ ⇒ err "incorrect argument number for if3; should be 3")
    ...
  )
```

```
(* Computes a value from expressions that might contain thunks. *)
and forceValue (v1: value, t1: typ): value * typ = ...

and evaluate (ex: exp, en: env): value * typ =
(
  ...
  case ex of
    Int_c i               ⇒ (Int_v i,    Int_t)
    ...
  | Id_e id               ⇒ (case lookupBinding (id, en) of
                               NONE      ⇒ err ("unbound variable "^id)
                             | SOME((Dyn_v ex, _))
                                         ⇒ evaluate(ex, en)
                             | SOME v    ⇒ v)
  | If_e (test, e1, e2)   ⇒
       (case forceValue (evaluate(test, en)) of
          (Bool_v b, Bool_t)⇒ evaluate(if b then e1 else e2, en)
        | _                 ⇒ err ("'if' condition must be boolean"))
  | Let_e   (dlist, ex)   ⇒ evaluate (ex, evaluateDeclare (dlist, en))
  | Apply_e (e1, e2)      ⇒ evaluateApply(e1, e2, en)
  | Unop_e  (uop, ex)     ⇒ evaluateUnop (uop,
                                 forceValue(evaluate(ex, en)))
  ...
)

and evaluateApply (e1: exp, e2: exp, encrt: env): value * typ =
(
  ...
  let
  (*fc = function, fa = formal arg, a = actual arg, t = type, l = list*)
    val (fc, fct)  = forceValue (evaluate(e1, encrt))
  in
    case fc of
      SpecForm_v name              ⇒ specialForm(name, e2, encrt)
    | Predef_v   name              ⇒ predefined (name,
                                          evaluate (e2, encrt))
    | Fn_v(fal, env, body, name)  ⇒
       let
        (* Are we using static or dynamic scoping? *)
        val en = case scoping of
                     STATIC  ⇒ env
                   | DYNAMIC ⇒ encrt
        (* Evaluate (maybe now, maybe later) the function's arguments *)
        val (a,  at)   =
          (case evaluation of
             EAGER ⇒ evaluate (e2, encrt)
           | LAZY  ⇒
               (case e2 of
                  Tuple_e a2 ⇒
                    (Tuple_v(map (fn a3 ⇒ Thunk_v(a3, encrt)) a2),
                     Tuple_t(map (fn a3 ⇒ Undef_t) a2))
                | _ ⇒ (Thunk_v(e2, encrt), Undef_t)))
        (* Transfer values & types of actual parameters into lists. *)
        val (al, atl) = case (a, at) of
               (Tuple_v al, Tuple_t atl) ⇒ (al,  atl)
             | (_,          _          ) ⇒ ([a], [at])
        (* First, retrieve the types of the formal arguments.    *)
        val (fatl, frt)  = case fct of
                          Fn_t(Tuple_t fatl, frt) ⇒ (fatl, frt)
                        | _ ⇒ err "internal err, bad function type"
        (*
           Are there too many, too few, or just enough args?
           (Length atl = Length al) & (Length fal = Length falt)
           1 = set of matched args
           2 = supplementary or missing args
        *)
        val (fal1, fatl1, al1, atl1, fal2, fatl2, al2, atl2) =
          let
            val len = Int.min(List.length atl, List.length fal)
          in
            (List.take(fal, len), List.take(fatl, len),
             List.take(al,  len), List.take(atl,  len),
             List.drop(fal, len), List.drop(fatl, len),
             List.drop(al,  len), List.drop(atl,  len))
          end

        (*
          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 ()
        (*
          Do types match for the available actual arguments?
          u = unified
        *)
        val utl = ListPair.map (fn (f, a) ⇒ unifyTypes(f, a))
                               (fatl1, atl1)
                  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(fal2,
                  ListPair.foldl (fn ((fa, a), ut, en') ⇒
                                     insertBinding(fa, (a, ut), en'))
                                 en
                                 (ListPair.zip(fal1, al1), utl),
                  body,
                  NONE),
             Fn_t(Tuple_t fatl2, frt))
          else
            (case evaluation of
              EAGER ⇒
                (*evaluate function, check returned type (r = returned)*)
                let
                  val (rv, rt) = evaluate(
                    body,
                    ListPair.foldl
                      (fn ((fa, a), ut, en') ⇒
                         insertBinding(fa, (a, ut), en'))
                      (case name of
                         NONE    ⇒ en
                       | SOME(s)  ⇒ insertBinding(s,(fc,fct),en))
                      (ListPair.zip(fal1, al1), utl))
                  val urt = unifyTypes(frt, rt)
                            handle TypeUnification ⇒
                              err ("actual return type does not" ^
                                   "match declared type")
                in
                  (rv, urt)
                end
            | LAZY  ⇒
               (* create thunk for fct body and unevaluated arguments *)
               (Thunk_v(
                  body,
                  ListPair.foldl
                    (fn ((fa, a), ut, en') ⇒
                       insertBinding(fa, (a, ut), en'))
                    (case name of
                       NONE    ⇒ en
                     | SOME(s) ⇒ insertBinding(s, (fc,fct), en))
                    (ListPair.zip(fal1, al1), utl)),
                frt))
       end
    | _  ⇒ err "attempt to evaluate non−function"
  end
)

and evaluateUnop (uop: unop, (v, t): value * typ): value * typ = ...

and evaluateBinop (bop: binop,
                   (v1, t1): value * typ,
                   (v2, t2): value * typ): value * typ =
(
  case (bop, v1, v2) of
    (Plus,  Int_v  a, Int_v  b)          ⇒ (Int_v  (a+b),     Int_t )
  | (Plus,  Real_v a, Real_v b)          ⇒ (Real_v (a+b),     Real_t)
  | (Plus,  _,        _       )          ⇒ err "type error (+)"
  ...
)
and evaluateDeclare (dlist: decl list, en: env): env = ...
end
```