```
open Pp

type valeur =
    | Vint of int
    | Vbool of bool
    | Varray of valeur array
    | Undefined

type erreur =
    | Inconnue of string
    | DivZero
    | Type of type_expr * valeur
    | NumArgs of string
    | ProcFun of string
    | TypeArray of valeur
    | Bound of int * int (* taille, indice *)
    | PasDef

exception Erreur of erreur

let erreur e = raise (Erreur e)
;;

(* Les environnements *)

type env_t = {
  fonctions : (string * definition) list ;
  globals : (string * valeur ref) list ;
  locals : (string * valeur ref) list
  }

let trouve_var {globals = glob ; locals = loc } x =
  try List.assoc x loc with
  | Not_found ->
      try List.assoc x glob with
      | Not_found -> erreur (Inconnue ("variable: "^x))

and trouve_fun {fonctions = funs } x =
  try List.assoc x funs with
  | Not_found ->
      erreur (Inconnue ("fonction: "^x))

(*
  Copie d'enregistrement avec extension
  Voir section 6.5 du manuel OCaml.
*)

and extend x v env =
  {env with locals = (x,ref v) :: env.locals}

and take_globs env =
  {env with locals = []}
;;


let binop op i1 i2 = match op with
| Plus -> Vint (i1 + i2)
| Minus -> Vint (i1 - i2)
| Times -> Vint (i1 * i2)
| Div ->
    if i2 = 0 then
      erreur DivZero
    else
      Vint (i1/i2)
| Lt -> Vbool (i1 < i2)
| Le -> Vbool (i1 <= i2)
| Gt -> Vbool (i1 > i2)
| Ge -> Vbool (i1 >= i2)
| Eq -> Vbool (i1 = i2)
| Ne -> Vbool (i1 <> i2)
```

```
    let check_bound t i =
      let taille = Array.length t in
75    if  i < 0 || i >= taille then
        erreur (Bound (taille, i))

    let rec expr env = function
    | Int i -> Vint i
80  | Bool b -> Vbool b
    | Bin (op,e1, e2) ->
        let i1 = expr_int env e1 in
        let i2 = expr_int env e2 in
        binop op i1 i2
85  | Get x -> !(trouve_var env x)
    | Function_call (f,e_args) ->
        let fdef = trouve_fun env f in
        let new_env =
          extend
90          f Undefined (* variable re'sultat *)
            (call_env env f fdef e_args) in
        instr_list new_env fdef.body ;
        let fcell = trouve_var new_env f in (* recuperer le re'sultat *)
        !fcell
95  | Geti (et, ei) ->
        let vt = expr_array env et in
        let vi = expr_int env ei in
        check_bound vt vi ;
        vt.(vi)
100 | Alloc (ei, t) ->
        let i = expr_int env ei in
        Varray (Array.make i Undefined)

    and expr_int env e = match expr env e with
105 | Vint i -> i
    | Undefined -> erreur PasDef
    | v -> erreur (Type (Integer,v))

    and expr_bool env e = match expr env e with
110 | Vbool b -> b
    | Undefined -> erreur PasDef
    | v -> erreur (Type (Boolean,v))

    and expr_array env e =  match expr env e with
115 | Varray a -> a
    | Undefined ->  erreur PasDef
    | v ->  erreur (TypeArray v)

    and env_args env f xs es = match xs,es with
120 | [],[] -> take_globs env
    | (x,t)::xs, e::es ->
        let v = expr env e in
        extend x v (env_args env f xs es)
    | _ -> erreur (NumArgs f)
125
    and env_locs env = function
    | [] -> env
    | (x,_) :: reste ->
          extend x Undefined (env_locs env reste)
130
    and call_env env f fdef es  =
      env_locs
        (env_args env f fdef.arguments es)
        fdef.local_vars
135
    and instr env = function
      | Set (x, e) ->
          let xcell = trouve_var env x in
          let v = expr env e in
140       xcell := v
      | Sequence is -> instr_list env is
      | If (eb,itrue,ifalse) ->
```

```
        if expr_bool env eb then
          instr env itrue
        else
          instr env ifalse
    | While (ec,i) ->
        while expr_bool  env ec do
          instr env i
        done
    | Procedure_call (f,e_args) ->
        let fdef = trouve_fun env f in
        let new_env = call_env env f fdef e_args in
        instr_list new_env fdef.body
    | Write_int e ->
        let i = expr_int env e in
        print_int i
    | Writeln_int e ->
        let i = expr_int env e in
        print_int i ;
        print_newline ()
    | Read_int x ->
        let xcell = trouve_var env x in
        xcell := Vint (read_int ())
    | Seti (et, ei, ev) ->
        let t = expr_array env et in
        let i = expr_int env ei in
        let v = expr env ev in
        check_bound t i ;
        t.(i) <- v

and instr_list env = function
    | [] -> ()
    | i::rem ->
        instr env i ; instr_list env rem

let eval
    {global_vars = globs ;
      definitions = defs ;
      main = i} =
  let start_env =
    {globals = List.map (fun (x,_) -> x, ref Undefined) globs ;
      fonctions  = defs ;
      locals = []} in
  try
    instr_list start_env i
  with
    | Erreur e ->
        begin match e with
        | Inconnue x ->
            prerr_endline ("Variable inconnue: "^x)
        | DivZero ->
            prerr_endline "Division par zéro"
        | Type (t,v) ->
            prerr_endline "Erreur de type"
        | NumArgs s ->
            prerr_endline ("Appel de fonction/procédure "^s)
        | ProcFun s ->
            prerr_endline (s^" n'est pas une procédure/fonction")
        | TypeArray v ->
            prerr_endline ("Tableau")
        | PasDef ->
            prerr_endline "Valeur indéfinie"
        | Bound (taille, indice) ->
            prerr_endline
              ("Accès hors limite dans un tableau de taille "^
               string_of_int taille ^ ":"^string_of_int indice)
        end ;
        exit (1)
```