```
open Pp
open Code

(* compilation des binop *)
let cbinop = function
  | Pp.Plus -> Plus
  | Pp.Minus -> Minus
  | Pp.Times -> Times
  | Pp.Div -> Div
  | Pp.Lt -> Lt
  | Pp.Le -> Le
  | Pp.Gt -> Gt
  | Pp.Ge -> Ge
  | Pp.Eq -> Eq
  | Pp.Ne -> Ne

(* idem pour les relop *)
exception NotRelop of string

let crelop = function
  | Pp.Lt -> Rlt
  | Pp.Le -> Rle
  | Pp.Gt -> Rgt
  | Pp.Ge -> Rge
  | Pp.Eq -> Req
  | Pp.Ne -> Rne
  | Pp.Plus -> raise (NotRelop "+")
  | Pp.Minus -> raise (NotRelop "-")
  | Pp.Times -> raise (NotRelop "*")
  | Pp.Div   -> raise (NotRelop "/")

(* Le type des informations associées aux variables dans
   l'environnement. *)

type access =
  | Local of Gen.temp    (* variable stockée dans un temporaire *)
  | Address of Code.exp (* variable stockée en mémoire à l'adresse donnée par cette expression *)

(* Lecture d'une variable. Le code engendré diffère selon que la
   variable est stockée dans un temporaire ou en mémoire. *)
let r_value = function
  | Local t -> Temp t
  | Address a -> Mem a

(* Écriture dans une variable. Même remarque. *)
let store a ec =
  match a with
  | Local r -> Move_temp (r, ec)
  | Address m -> Move_mem (m, ec)

(* expressions *)
let rec cexpr env (e : Pp.expression) : Code.exp =
  match e with
  | Int i  ->
      Const i
  | Bool true ->
      Const 1
  | Bool false ->
      Const 0
  | Get s ->
      r_value (Env.find_var env s)
  | Pp.Bin (op, e1, e2) ->
      Bin (cbinop op, cexpr env e1, cexpr env e2)
  | Function_call (f, args) ->
      Call (Env.find_definition env f, List.map (cexpr env) args)
  | Geti (e1,e2) ->
      Mem (cindex env e1 e2)
  | Alloc (e,_) ->
      Call (Frame.alloc, [cexpr env e])

```

```
(* indexation d'un tableau *)
and cindex env e1 e2 =
  let offset =
    match cexpr env e2 with
    | Const i2 -> Const (Frame.wordsize * i2) (* petite optimisation... *)
    | ce2 -> Bin (Times, Const Frame.wordsize, ce2)
  in
  Bin (Uplus, cexpr env e1, offset)

let rec cinstruction env (instruction : Pp.instruction) : Code.stm =
  match instruction with
  | Set (s,e) ->
      store (Env.find_var env s) (cexpr env e)
  | Sequence is ->
      Seq (cinstruction_list env is)
  | If (e,inst,insf) ->
      begin match e with
        Pp.Bin (op,e1,e2) ->
          ccond env op e1 e2 inst insf
      | _ ->
          ccond env Pp.Ne e (Bool false) inst insf
      end
  | While (e,i) ->
      begin match e with
        Pp.Bin (op,e1,e2) ->
          cwhile env op e1 e2 i
      | _ ->
          cwhile env Pp.Ne e (Bool false) i
      end
  | Procedure_call (f,args) ->
      Exp (Call (Env.find_definition env f,List.map (cexpr env) args))
  | Write_int e ->
      Exp (Call (Frame.write_int, [cexpr env e]))
  | Writeln_int e ->
      Exp (Call (Frame.writeln_int, [cexpr env e]))
  | Read_int s ->
      store (Env.find_var env s) (Call (Frame.read_int,[]))
  | Seti (e1, e2, e3) ->
      Move_mem (cindex env e1 e2, cexpr env e3)

and cinstruction_list env (is : Pp.instruction list) : Code.stm list =
  List.map (cinstruction env) is


(* compilation de la conditionnelle ``If'' *)
and ccond env op e1 e2 inst insf =
  let lt = Gen.new_label ()
  and lf = Gen.new_label ()
  and lend = Gen.new_label () in
  Seq
    [Cjump (crelop op,cexpr env e1, cexpr env e2,lt,lf) ;
     Label lf ;
     cinstruction env insf ;
     Jump lend ; Label lt ;
     cinstruction env inst ;
     Label lend]

(* compilation de la boucle  ``While'' *)
and cwhile env op e1 e2 i =
  let enter = Gen.new_label ()
  and test = Gen.new_label ()
  and lout = Gen.new_label () in
  Seq
    [Jump test ;
     Label enter ;
     cinstruction env i ;
     Label test ;
     Cjump (crelop op,cexpr env e1,cexpr env e2,enter,lout) ;
     Label lout]
```

```
type 'a procedure = Frame.frame * 'a

let cfun env (fundef : string * Pp.definition) : Code.stm procedure =
  let s, {arguments = args ; result = r ; local_vars = locs ; body = ins} = fund
ef in

  (* Commençons par obtenir le frame de la fonction. Celui-ci a déjà
     été construit et stocké dans l'environnement [env]. *)

  let f = Env.find_definition env s in

  (* Construire la portion d'environnement correspondant
     aux arguments de la fonction. On connaît leurs noms
     grâce à la liste [args]. On peut consulter le frame
     [f] pour obtenir les temporaires qui leurs sont
     associés. Reste à construire une liste de paires
     associant noms et temporaires. *)
  let env_args =
    List.fold_right2 (fun (nom, _) temporaire accu ->
      (nom, Local temporaire) :: accu
    ) args (Frame.frame_args f) [] in

  (* Construire la portion d'environnement correspondant
     au résultat de la fonction. Son nom est celui de la
     fonction. On consulte le frame pour connaître le
     temporaire associé. *)
  let env_result =
    match Frame.frame_result f with
    | None ->
        []
    | Some temporaire ->
        [ (s, Local temporaire) ] in

  (* Enfin, allouons des temporaires pour représenter les
     variables locales, et construisons la portion
     d'environnement correspondante. *)
  let env_locs =
    List.map (fun (nom, _) ->
      let temporaire = Gen.new_temp() in
      (nom, Local temporaire)
    ) locs in

  (* Reste à modifier l'environnement fourni en y installant
     les trois portions ci-dessus. *)
  let new_env =
    Env.change_local_vars env (env_args @ env_result @ env_locs) in

  (* On renvoie le frame et la séquence d'instructions correspondant
     au corps de la fonction. Celle-ci est construite à l'aide du
     nouvel environnement. *)
  f, Seq (cinstruction_list new_env ins)

 (* Définition du type des programmes. *)

type 'a program =
    { number_of_globals : int;
      main : 'a procedure;
      procedures : 'a procedure list
    }

 (* Cette fonction auxiliaire alloue les variables globales.
    Elle utilise [glob_offset] pour mémoriser l'offset de la
    prochaine variable globale disponible. *)

let glob_offset =
  ref 0

let make_global () =
  let offset = !glob_offset in
  glob_offset := offset + Frame.wordsize;
  if offset = 0 then
```

```
      Address (Temp Frame.global_register)
     else
      Address (Bin (Uplus, Temp Frame.global_register, Const offset))

let cprog (prog : Pp.program) : Code.stm program =
  let {global_vars = g ; definitions = defs ; Pp.main = p} = prog in

  (* Allouer les variables globales. *)
  let globals =
    List.map
      (fun (x, _) -> (x, make_global()))
    g in

  (* Faire passer le code principal pour une procédure. *)
  let main_def = ("main", {arguments = [] ; result = None ; local_vars = [] ; bo
dy = p}) in

  (* Construire des frames pour les fonctions/procédures
     et pour le programme principal. Construire ensuite
     l'environnement initial. *)

  let make_frame (s, {arguments = args ; result = r }) =
    s, Frame.named_frame s args r in

  let env_init =
    Env.create_global
      globals
      (List.map make_frame (main_def :: defs)) in

  (* Compiler les procédures/fonctions et le programme
     principal dans cet environnement. *)

  {
    number_of_globals = List.length globals ;
    main = cfun env_init main_def ;
    procedures = List.map (cfun env_init) defs
  }

let program p =
try
  cprog p
with
| Env.Free s ->
    Printf.fprintf stderr
      "Erreur interne:\n\
  la variable %s est libre" s;
      prerr_newline(); exit 1
| NotRelop op ->
    Printf.fprintf stderr
      "Erreur interne:\n\
  l'opérateur %s est utlisé comme une relation" op ;
      prerr_newline(); exit 1
```