```

(* les deux types de flux utilisés: le flux à parser et le flux des solutions *)
(* (le fait de passer () à Make assure que ces deux types de flux seront différents et ne pourront donc pas être mélangés involontairement) *)
module Flux = Monadic_flux.Make ();;
module Solution = Monadic_flux.Make ();;

(* types des parsers généraux *)
type ('a, 'b) result = ('b * 'a Flux.t) Solution.t;;
type ('a, 'b) parser = 'a Flux.t -> ('a, 'b) result;;

(* interface des parsers: combinateurs de parsers et parsers simples *)
module type Parsing =
  sig
    val map : ('b -> 'c) -> ('a, 'b) parser -> ('a, 'c) parser

    val return : 'b -> ('a, 'b) parser

    val ( >>= ) : ('a, 'b) parser -> ('b -> ('a, 'c) parser) -> ('a, 'c) parser

    val zero : ('a, 'b) parser

    val ( ++ ) : ('a, 'b) parser -> ('a, 'b) parser -> ('a, 'b) parser

    val run : ('a, 'b) parser -> 'a Flux.t -> 'b Solution.t

    val pvide : ('a, unit) parser

    val ptest : ('a -> bool) -> ('a, 'a) parser

    val ( *> ) : ('a, 'b) parser -> ('a, 'c) parser -> ('a, 'b * 'c) parser
  end

(* implantation des parsers, comme vu en TD. On utilise les opérations *)
(* du module Flux et du module Solution                                *)
module Parser : Parsing =
  struct
    let map fmap parse f = Solution.map (fun (b, f') -> (fmap b, f')) (parse f);;

    let return b f = Solution.return (b, f);;

    let (>>=) parse dep_parse = fun f -> Solution.(parse f >>= fun (b, f') -> dep_parse b f');;

    let zero f = Solution.zero;;

    let (++) parse1 parse2 = fun f -> Solution.(parse1 f ++ parse2 f);;

    let run parse f = Solution.(map fst (filter (fun (b, f') -> Flux.uncons f' = None) (parse f)));;

    let pvide f =
      match Flux.uncons f with
      | None   -> Solution.return ((), f)
      | Some _ -> Solution.zero;;

    let ptest p f =
      match Flux.uncons f with
      | None        -> Solution.zero
      | Some (t, q) -> if p t then Solution.return (t, q) else Solution.zero;;

    let ( *> ) parse1 parse2 = fun f ->
      Solution.(parse1 f >>= fun (b, f') -> parse2 f' >>= fun (c, f'') -> return ((b, c), f''));;
  end

(* Le type des programmes LOGO *)
type prog = decls * inst
and decls = decl list
and decl = Decl of string * prog
and inst = cmd list
(* Le type des commandes LOGO  *)
and cmd =
| Repeat of int * prog      (* on repete n fois un programme *)
| Move of int               (* on se deplace de d pas dans la direction courante  *)
| Turn of int               (* on tourne d'un angle a *)
| On                        (* on pose le stylo sur la feuille *)
| Off                       (* on leve le stylo *)
| Call of string            (* on appelle un sous-progremme *)


open Parser

(* Parsers pour les terminaux du langege LOGO.            *)
(* L'analyse lexicale est realisee à la main              *)
(* sur des flux de caracteres sans lexer externe          *)
(* Les parsers de mots-clés renvoient (), de type unit    *)
(* Le parser des entiers renvoie la valeur lue            *)

(* 'droppe' le resultat d'un parser et le remplace par () *)
let drop p = map (fun x -> ()) p;;

(* Parser pour les espaces, retour-chariots, tabulation   *)
let is_space c = String.contains " \t\r\n" c;;
let space  = drop (ptest is_space);;

(* Combinateur de parser qui consomme tous les espaces    *)
(* avant d'appeler le parser 'p'                          *)
let rec eat_space p flux =
  (map snd (space *> (eat_space p)) ++ p) flux;;

(* Parser qui reconnait le caractere 'c'                  *)
let p_car c = drop (ptest ((=) c));;
(* Parser qui reconnait la chaine 's'                     *)
let p_chaine s =
  let rec parse i =
    if i < 0
    then return ()
    else map fst (parse (i-1) *> p_car s.[i])
```

```
  in parse (String.length s - 1)


(* ****************************************************** *)
(* Parsers pour la fin de fichier                         *)
(* ****************************************************** *)

let p_eof = eat_space pvide;;


(* ****************************************************** *)
(* Parsers pour les mots-clés                             *)
(* ****************************************************** *)

let p_ptvirg = eat_space (p_car ';');;
let p_begin  = eat_space (p_chaine "begin");;
let p_end    = eat_space (p_chaine "end");;
let p_repeat = eat_space (p_chaine "repeat");;
let p_move   = eat_space (p_chaine "move");;
let p_turn   = eat_space (p_chaine "turn");;
let p_on     = eat_space (p_chaine "on");;
let p_off    = eat_space (p_chaine "off");;

let p_proc   = eat_space (p_chaine "proc");;
let p_call   = eat_space (p_chaine "call");;

(* ****************************************************** *)
(* Parser pour les constantes entieres                    *)
(* ****************************************************** *)

(* Parser pour les chiffres                               *)
let is_chiffre c = String.contains "0123456789" c;;
let p_chiffre = ptest is_chiffre;;

let p_entier =
  let rec horner acc =
    p_chiffre >>= fun c -> let acc' = 10 * acc + (Char.code c - Char.code
'0') in horner acc' ++ return acc'
  in eat_space (horner 0);;


(* ****************************************************** *)
(* Parser pour les identificateurs                        *)
(* ****************************************************** *)

let is_lettre c = c >= 'a' && c <= 'z';;
let p_lettre = ptest is_lettre;;

let p_ident =
  let rec concat acc =
    p_lettre >>= fun l -> concat (l::acc) ++ return (l::acc)
  in map (fun liste -> String.init (List.length liste) (List.nth (List.rev
liste)))
         (eat_space (concat []));;

(* Grammaire LL1 des programmes LOGO:
 P -> begin D I end
 D -> /\
 D -> S ; D
 S -> proc ident P
 I -> /\
 I -> C ; I
 C -> repeat entier P
 C -> move entier
 C -> turn entier
 C -> on
 C -> off
 *)


(* les parsers mutuellement récursifs pour la grammaire ci-dessus: à
compléter *)
let rec parse_P : (char, prog) parser = fun flux ->
  (*Format.printf "parse_P@ ";*)
  (
  (*autre solution avec map et *>
    map (fun (((_, decls), inst), _) -> (decls, inst)) (p_begin *> parse_D *>
parse_I *> p_end)
   *)
    p_begin >>= fun () -> parse_D >>= fun decls -> parse_I >>= fun inst ->
p_end >>= fun () -> return (decls, inst)
  ) flux
and parse_I : (char, inst) parser = fun flux ->
  (*Format.printf "parse_I@ ";*)
  (
      (return [])
    ++
  (*autre solution avec map et *>
    map (fun ((c, _), i) -> c::i) (parse_C *> p_ptvirg *> parse_I)
   *)
      (parse_C >>= fun cmd -> p_ptvirg >>= fun () -> parse_I >>= fun inst ->
return (cmd::inst))
  ) flux
and parse_C : (char, cmd) parser = fun flux ->
  (*Format.printf "parse_C@ ";*)
  ((* autre solution avec map et *>
    map (fun ((_, n), p) -> (Repeat (n, p))) (p_repeat *> p_entier *>
parse_P) ++
    map (fun (_, d) -> Move d) (p_move *> p_entier) ++
    map (fun (_, a) -> Turn a) (p_turn *> p_entier) ++
    map (fun _ -> On) p_on ++
    map (fun _ -> Off) p_off ++
    map (fun (_, id) -> Call id) (p_call *> p_ident)
    *)
     (p_repeat >>= fun () -> p_entier >>= fun n -> parse_P >>= fun prog ->
return (Repeat (n, prog)))
   ++
     (p_move >>= fun () -> p_entier >>= fun n -> return (Move n))
   ++
     (p_turn >>= fun () -> p_entier >>= fun n -> return (Turn n))
   ++
     (p_on >>= fun () -> return On)
   ++
     (p_off >>= fun () -> return Off)
   ++
     (p_call >>= fun () -> p_ident >>= fun ident -> return (Call ident))
  ) flux
and parse_D : (char, decls) parser = fun flux ->
  (*Format.printf "parse_D@ ";*)
  (
```

```
      (return [])
    ++
  (*autre solution avec map et *>
    map (fun ((s, _), d) -> s::d) (parse_S *> p_ptvirg *> parse_D)
   *)
      (parse_S >>= fun decl -> p_ptvirg >>= fun () -> parse_D >>= fun decls -
> return (decl::decls))
  ) flux
and parse_S : (char, decl) parser = fun flux ->
  (*Format.printf "parse_S@ ";*)
  (
  (*autre solution avec map et *>
    map (fun ((_, id), p) -> Decl (id, p)) (p_proc *> p_ident *> parse_P)
   *)
    p_proc >>= fun () -> p_ident >>= fun ident -> parse_P >>= fun prog ->
return (Decl (ident, prog))
  ) flux


(* fonction principale de parsing des programmes LOGO *)
let parse_logo flux = run (map fst (parse_P *> p_eof)) flux;;


(* fonctions auxiliaires *)
(* flux construit à partir des caractères de la chaîne s *)
let flux_of_string s =
  Flux.unfold (fun (i, l) -> if i = l then None else Some (s.[i], (i+1, l)))
(0, String.length s);;

(* flux construit à partir du contenu du fichier 'name' *)
let flux_of_file name =
  let f = open_in name in
  Flux.unfold (fun () -> try Some (input_char f, ()) with End_of_file ->
close_in f; None) ();;

(* conversion d'un programme en chaîne de caractères *)
let rec logo_to_string (d, p) =
  Format.sprintf "begin %s %s end"
    (String.concat " " (List.map (fun (Decl (id, p)) -> Format.sprintf "proc
%s %s" id (logo_to_string p)) d))
    (String.concat " " (List.map (fun c -> Format.sprintf "%s;"
(cmd_to_string c)) p))
and cmd_to_string c =
  match c with
  | Repeat (n, p) -> Format.sprintf "repeat %d %s" n (logo_to_string p)
  | Move   d      -> Format.sprintf "move %d" d
  | Turn   a      -> Format.sprintf "turn %d" a
  | On            -> Format.sprintf "on"
  | Off           -> Format.sprintf "off"
  | Call id       -> Format.sprintf "call %s" id

(* affichage des programmes LOGO solutions du parsing *)
let rec print_solutions progs =
  match Solution.uncons progs with
  | None        -> ()
  | Some (p, q) ->
     begin
       Format.printf "LOGO program recognized: %s@." (logo_to_string p);
       print_solutions q
     end;;


(* programme interactif de test qui parse un programme LOGO lu au clavier *)
(* puis affiche tous les parsings possibles                               *)
let test_parser_logo () =
  let rec loop () =
    Format.printf "programme?@.";
    flush stdout;
    let l = read_line () in
    let f = flux_of_string l in
    let progs = parse_logo f in
    match Solution.uncons progs with
    | None        -> (Format.printf "** parsing failed ! **@."; loop ())
    | Some (p, q) ->
       begin
         print_solutions (Solution.cons p q);
         loop ()
       end
  in loop ();;

let rad_of_deg = 2. *. Float.pi /. 360.

let rec exec_logo (on, x, y, a, st) (decls, inst) =
  let st' = List.map (fun (Decl (id, p)) -> (id, p)) decls in
  List.fold_left exec_cmd (on, x, y, a, List.rev_append st' st) inst
and exec_cmd (on, x, y, a, st) cmd =
  match cmd with
  | Repeat (n, p)   -> if n <= 0 then (on, x, y, a, st) else exec_cmd
(exec_logo (on, x, y, a, st) p) (Repeat (n-1, p))
  | Move d          -> let x' = x +. float_of_int d *. cos (rad_of_deg *. a)
                       and y' = y +. float_of_int d *. sin (rad_of_deg *. a)
                       in begin
                           (if on then Graphics.lineto else Graphics.moveto)
(int_of_float x') (int_of_float y');
                           (on, x', y', a, st)
                         end
  | Turn b          -> (on, x, y, mod_float (a +. (float_of_int b)) 360., st)
  | On              -> (true , x, y, a, st)
  | Off             -> (false, x, y, a, st)
  | Call id         -> let (on', x', y', a', _) = exec_logo (on, x, y, a, st)
(List.assoc id st)
                       in (on', x', y', a', st)

let run_logo prog =
  begin
    Graphics.open_graph " 800*600";
    Graphics.moveto 400 300;
    ignore (exec_logo (false, 400., 300., 0., []) prog);
    ignore (read_line ());
    Graphics.close_graph ()
  end

let _ =
  let f = flux_of_string "begin proc segment begin move 20; turn 60; end; on;
repeat 6 begin call segment; end; end" in
  let p = parse_logo f in
    match Solution.uncons p with
    | None        -> assert false
    | Some (p, _) -> (Format.printf "%s@ " (logo_to_string p); run_logo p)
```