```
(**********************************************************************)
(*                              Min Max                               *)
(**********************************************************************)

(* minimize : ('a -> 'a -> bool) -> ('b -> 'a) -> 'b list -> 'b * 'a
 *
 * prend une fonction de comparaison, une fonction d'evaluation, une liste,
 * et rend un element de la liste minimizant la fonction d'evaluation,
 * ainsi que la valeur de la fonction d'evaluation en ce point. *)

let minimize comp f =
  function
      [] -> raise (Invalid_argument "minimize: empty list")
    | e :: r ->
        it_list (fun ((_,m) as res) e ->
          let r = f e in
          if comp r m then (e,r) else res)
          (e,(f e)) r
;;

let minimize cmp eval l =
  let rec trouve ((m,v) as c) = function
      []    -> c
    | x'::l -> let v' = eval x' in
               if cmp v' v then
                 trouve (x',v') l
               else
                 trouve c l
  in match l with
      []   -> raise (Invalid_argument "minimize: empty list")
    | x::l -> trouve (x,eval x) l
;;

(* min et max dans le cas particulier d'evaluation dans float *)

let min (eval:'a->float) l = minimize (prefix <) eval l;;
let max (eval:'a->float) l = minimize (prefix >) eval l;;


(* minmax
     eval : 'a -> float
     suiv : 'a -> 'a list
     dep  : 'a
   renvoie le coup à jouer et son evaluation : 'a (pos. choisie) * float
   profondeur 1. *)

let minmax eval suiv dep =
  let eval1 p = match eval p with
      0.0 -> 1.0
    | 1.0 -> 0.0
    | _   -> snd (min eval (suiv p))
  in
    max eval1 (suiv dep)
;;


(* minmax_n : min-max profondeur n *)

let minmax_n eval suiv n dep =

  let rec eval1 n p = match eval p with
      0.0 -> 1.0
```



```
      | 1.0 -> 0.0
      | _   ->
          if n = 0 then
            snd (min eval (suiv p))
          else
            snd (min (fun p -> snd (minmax_rec (pred n) p)) (suiv p))

  and minmax_rec n p =
    match eval p with
        (0.0 | 1.0) as e -> p,e
      | _                -> max (eval1 n) (suiv p)

  in minmax_rec n dep
;;

(* remarque : minmax = (minmax_n 0) *)

(**********************************************************************)
(*                         Jeu des allumettes                         *)
(**********************************************************************)

type configuration == bool * int ;;

let (configuration_initiale : configuration) = true, 20 ;;

let (suiv : configuration -> configuration list) = fun (b,n) ->
  map (fun n' -> not b,n') (match n with
      0 -> []
    | 1 -> [0]
    | 2 -> [0 ; 1]
    | n -> [n-3 ; n-2 ; n-1])
;;

let (eval : configuration -> float) = fun (b,n) -> match n with
      0 -> 1.0
    | _ -> 0.5
;;

let joue ((b,n) as c) = fst (minmax_n eval suiv (n/2+1) c) ;;

let affiche_configuration (b,n) =
  print_string (if b then "au joueur 1 : " else "au joueur 2 : ") ;
  for i = 1 to n do
    print_string "|"
  done ;
  print_newline ()
;;

let boucle coup1 coup2 ci =
  let rec boucle_rec ((b,n) as c) =
    if n = 0 then begin
      print_string (if b then "le joueur 1 gagne !"
                    else "le joueur 2 gagne !") ;
      print_newline ()
    end else begin
      affiche_configuration c ;
      boucle_rec (if b then coup1 c else coup2 c)
    end
  in
    boucle_rec ci
;;
```

```
let rec joueur ((b,n) as c) =
  print_string "Combien d'allumettes prenez-vous ? ";
  let i = read_int () in
  if i <= n & 0 < i & i <= 3 then
    not b, (n-i)
  else
    joueur c
;;

boucle joue joueur configuration_initiale;;
boucle joueur (fun c -> let (Some c) = joueab c in c) configuration_initiale;;

(* alpha-beta *)

let rec max_list eval ((ea,alpha) as a) beta = function
    [] -> a
  | e :: r ->
      let v = eval alpha beta e in
      if v >= beta then (e,v)
      else if v > alpha then
        max_list eval (e,v) beta r
      else
        max_list eval a beta r
;;

let rec min_list eval alpha ((eb,beta) as b) = function
    [] -> b
  | e :: r ->
      let v = eval alpha beta e in
      if v <= alpha then (e,v)
      else if v < beta then
        min_list eval alpha (e,v) r
      else
        min_list eval alpha b r
;;

let rec alphabeta eval suiv alpha beta c =
  let eval1 alpha beta e = snd (min_list (fun _ _ e -> eval e)
                                         alpha (None,beta) (suiv e)) in
  max_list eval1 (None, alpha) beta (suiv c)
;;


let alphabeta_n eval suiv n c =
  let rec eval1 n alpha beta (Some p) =
    match eval p with
      0.0 -> 1.0
    | 1.0 ->
        (* printf__printf "eval1 %d, %f, %f, %d => 0.\n" n alpha beta (snd p); *
)
        0.0
    | _   ->
        if n = 0 then
          let res =
          snd (min_list (fun _ _ (Some e) -> eval e) alpha (None,beta) (suiv p))
          in
(*        printf__printf "eval1 %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" res; *)
          res
        else
          let res =
          snd (min_list (fun a b p -> snd (alphabeta_rec (n-1) a b p))
```



```
                    alpha (None,beta) (suiv p))
          in
(*        printf__printf "eval1 %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" res; *)
          res
  and alphabeta_rec n alpha beta (Some p) =
    match eval p with
        (0.0 | 1.0) as e ->
(*        printf__printf "Alphabeta_rec %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" e; *)
          (Some p),e
      | _               ->
          let res =
            max_list (eval1 n) (None, alpha) beta (suiv p)
          in let (_,r) = res in
(*        printf__printf "Alphabeta_rec %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" r; *)
          res
  in alphabeta_rec n (-0.1) 1.1 c
;;

let suiv2 c = map (fun x -> Some x) (suiv c);;
let joueab ((b,n) as c) = (alphabeta_n eval suiv (n/2+1) (c)) ;;

let i = 5 in
joueab (true,i),
joue (true,i);;
joueab configuration_initiale;;
joue configuration_initiale;;

(* alpha-beta 2 *)
let max_list eval alpha beta l =
  let rec aux (ea,alpha as a) beta = function
      [] -> a
    | e :: r ->
        let v = eval alpha beta e in
        if v >= beta then (Some e,v)
        else if v > alpha then
          aux (Some e,v) beta r
        else
          aux a beta r
  in aux (None,alpha) beta l
    ;;

let min_list eval alpha beta l =
  let rec aux alpha ((eb,beta) as b) = function
     [] -> b
   | e :: r ->
       let v = eval alpha beta e in
       if v <= alpha then (Some e,v)
       else if v < beta then
         aux alpha (Some e,v) r
       else
         aux alpha b r
  in aux alpha (None,beta) l
    ;;


let alphabeta_n eval suiv n c =
  let rec eval1 n alpha beta p =
    match eval p with
      0.0 -> 1.0
```

```
    | 1.0 ->
        (* printf__printf "eval1 %d, %f, %f, %d => 0.\n" n alpha beta (snd p); *
)
        0.0
    | _   ->
        if n = 0 then
          let res =
          snd (min_list (fun _ _ e -> eval e) alpha beta (suiv p))
          in
(*        printf__printf "eval1 %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" res; *)
          res
        else
          let res =
          snd (min_list (fun a b p -> snd (alphabeta_rec (n-1) a b p))
                 alpha beta (suiv p))
          in
(*        printf__printf "eval1 %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" res; *)
          res
  and alphabeta_rec n alpha beta p =
    match eval p with
        (0.0 | 1.0) as e ->
(*        printf__printf "Alphabeta_rec %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" e; *)
          Some p,e
      | _             ->
          let res = max_list (eval1 n) alpha beta (suiv p) in
(*        let (_,r) = res in
          printf__printf "Alphabeta_rec %d, %f, %f, %d" n alpha beta (snd p);
          printf__printf " => %f\n" r; *)
          res
  in
  match alphabeta_rec n (0.) 1. c with
  | None,f -> (hd (suiv c)),f
  | Some r,f -> r,f
;;

(* morpion *)

let ci = true, (make_vect 3 [|0;0;0|]);;

let sum v =
  let rec aux acc = function
    | 0 -> acc
    | n -> aux (acc+v.(n)) (pred n)
  in aux 0 (vect_length v - 1) ;;

let eval (b,t) =
  let t = if b then map_vect (map_vect (fun x -> -x)) t else t in
  let l1 = sum t.(0) and l2 = sum t.(1) and l3 = sum t.(2) in
  let c1 = sum (init_vect 3 (fun i -> t.(i).(0)))
  and c2 = sum (init_vect 3 (fun i -> t.(i).(1)))
  and c3 = sum (init_vect 3 (fun i -> t.(i).(2))) in
  let d1 = sum (init_vect 3 (fun i -> t.(i).(i)))
  and d2 = sum (init_vect 3 (fun i -> t.(i).(2-i))) in
  if (c1 = 3) or (c2 = 3) or (c3 = 3) or (d1 = 3) or (d2 = 3) or (l1 = 3) or (l2
 = 3) or (l3 = 3) then
    1.0
  else if (c1 = (-3)) or (c2 = (-3)) or (c3 = (-3)) or (d1 = (-3)) or (d2 = (-3)
) or (l1 = (-3)) or (l2 = (-3)) or (l3 = (-3)) then
    0.
```



```
  else
    let nbz = ref 0 in
    for i = 0 to 2 do
      for j = 0 to 2 do
        if t.(i).(j) = 0 then nbz := !nbz + 1
      done
    done;
    if !nbz > 0 then 0.5 else 0.8
;;

let pos = [|[|-1; 1; 0|]; [|-1; 1; 0|]; [|0; 0; -1|]|];;
init_vect 3 (fun i -> pos.(i).(2-i));;
eval (ci);;

let suiv (b,p) =
  let n = if b then 1 else -1 in
  let r = ref [] in
  for i = 0 to 2 do
    for j = 0 to 2 do
      if p.(i).(j) = 0 then
        let v = init_vect 3 (fun i -> copy_vect p.(i)) in
        v.(i).(j) <- n;
        r := (not b,v) :: !r
    done
  done;
  !r
;;

suiv (true,pos);;
let suiv2 c = map (fun x -> Some x) (suiv c);;
let joueab c = fst (alphabeta_n eval suiv (6) (c)) ;;

joueab ci;;

type 'a option = None | Some of 'a;;
```