TD06.ml

```
(*************************************)
(* Ensembles représentés par des listes *)
(*************************************)


(* Exercice 1 *)

(* Ces deux fonctions s'expriment très simplement à l'aide d'un fold,
   mais le calcul se poursuit alors systématiquement jusqu'au bout
   même quand on aurait pu répondre sans examiner tous les éléments. *)
let rec pour_tout pred u =
  match u with
  | [] -> true
  | x :: xs -> pred x && pour_tout pred xs

let rec existe pred u =
  match u with
  | [] -> false
  | x :: xs -> pred x || existe pred xs

let rec filtre pred u =
  match u with
  | [] -> []
  | x :: xs -> if pred x then x :: filtre pred xs else filtre pred xs


(* Exercice 2 *)

(* cette version n'arrête pas la recherche quand elle devient sans
   espoir, ce qui n'est pas très grave *)
let appartient u x = existe (( = ) x) u

(* cette version s'arrête dès que possible *)
let rec appartient_bis u x =
  match u with
  | [] -> false
  | y :: ys  -> (x = y) || ((x > y) && appartient_bis ys x)

let inclus u v = pour_tout (appartient v) u

let inter u v = filtre (appartient u) v

let rec egal u v =
  match u, v with
  | [], [] -> true
  | x :: xs, y :: ys -> x = y && egal xs ys
  | _ -> false

(* évidemment, il est plus simple d'écrire
   let egal = ( = )
   (et cela revient exactement au même.) *)

(* Exercice 3 *)

let rec union u v =
  match u, v with
  | [], _ -> v
  | _, [] -> u
  | x :: xs, y :: ys when x < y -> x :: union xs v
  | x :: xs, y :: ys when x > y -> y :: union u ys
  | x :: xs, y :: ys -> x :: union xs ys

let rec inter_eff u v =
  match u, v with
  | [], _ -> []
  | _, [] -> []
  | x :: xs, y :: ys when x < y -> inter_eff xs v
  | x :: xs, y :: ys when x > y -> inter_eff u ys
  | x :: xs, y :: ys -> x :: inter_eff xs ys

let rec inclus_eff u v =
  match u, v with
  | [], _ -> true
  | x :: xs, y :: ys when x = y -> inclus xs ys
  | x :: xs, y :: ys when x > y -> inclus u ys
  | _ -> false

(*************************)
(* Un peu de combinatoire *)
(*************************)

let map_prefixe x = List.map (fun u -> x :: u)


(* Exercice 4 *)

let rec parties u =
  match u with
  | [] -> [[]]
  | x :: xs -> let sans_x = parties xs in
               let avec_x = map_prefixe x sans_x in
               sans_x @ avec_x


(* Exercice 5 *)

let rec combinaisons u n =
  match u, n with
  | _, 0 -> [[]]
  | [], _ -> []
  | x :: xs, _ -> let sans_x = combinaisons xs n in
                  let avec_x = map_prefixe x (combinaisons xs (n - 1)) in
                  sans_x @ avec_x

let rec avec_repetitions u n =
  match u, n with
  | _, 0 -> [[]]
  | [], _ -> []
  | x :: xs, _ -> let sans_x = avec_repetitions xs n in
                  let avec_x = map_prefixe x (avec_repetitions u (n - 1)) in
                  avec_x @ sans_x

let parties_bis u =
  let rec ( <|> ) a b = if a >= b then [] else a :: (a + 1 <|> b) in
  List.flatten (List.map (combinaisons u) (0 <|> List.length u + 1))

(* Exercice 6 *)

let rec insertions x u =
    match u with
    | [] -> [[x]]
    | y :: ys -> (x :: u) :: (map_prefixe y (insertions x ys))

let rec applatit u =
  match u with
  | [] -> []
  | x :: xs -> x @ applatit xs

(*
   Autre possibilité :
   let applatit = List.fold_left ( @ ) []

   Cette fonction est prédéfinie, c'est List.flatten.
*)
```

TD06.ml

```
let rec permutations u =
  match u with
  | [] -> [[]]
  | x :: xs -> List.flatten (List.map (insertions x) (permutations xs))

(* Exercice 7 *)

let rec plistes u p =
  if p = 0 then
    [[]]
  else
    let moins_un = plistes u (p - 1) in
    List.flatten (List.map (fun x -> map_prefixe x moins_un) u)


(****************************************)
(* Tableaux représentant des permutations *)
(****************************************)

let estPermutation t =
  let n = Array.length t in
  let occ = Array.create n 0 in
  let res = ref true in
  for i = 0 to n - 1 do
    if t.(i) >= 0 && t.(i) <= n - 1 then occ.(t.(i)) <- occ.(t.(i)) + 1
  done;
  for i=0 to n - 1 do
    res := !res && (occ.(i) = 1)
  done;
  !res

let _ =
  assert (estPermutation [|2; 1; 3; 0; 4|]);
  assert (not (estPermutation [|7; 1; 3; 0; 4|]));
  assert (not (estPermutation [|3; 1; 3; 0; 4|]));
  assert (not (estPermutation [|2; 3; 4; 5|]))

(*
   Toutes les fonctions suivantes gagneraient à être ré-écrites en
   "if estPermutation t then ... else failwith ...", je ne l'ai pas
   fait par souci de concision.
*)

let composer t u =
  let n = Array.length t in
  let v = Array.make n 0 in
  for i = 0 to n - 1 do
    v.(i) <- t.(u.(i))
  done;
  v

let _ = assert (composer [|1; 2; 0; 3|] [|3; 1; 2; 0|] = [|3; 2; 0; 1|])


let inverser t =
  let n = Array.length t in
  let u = Array.make n 0 in
  for i = 0 to n - 1 do
    u.(t.(i)) <- i
  done;
  u

let id n = Array.init n (fun x -> x);;

let _ = assert (composer [|1;0;3;4;2;5|] (inverser [|1;0;3;4;2;5|]) = id 6)
```

```
(* Calcul de puissance par exponentiation rapide. *)
let puis t k =
  let rec aux t k =
    match k, k mod 2 with
        (0, _) -> id (Array.length t)
      | (_, 0) -> let tmp = aux t (k / 2) in composer tmp tmp
      | _ -> composer t (aux t (k - 1))
  in
  if k < 0 then aux (inverser t) (-k) else aux t k

let _ = assert (puis [|1; 2; 3; 4; 5; 0|] 5 = [|5; 0; 1; 2; 3; 4|])

(* Pour calculer l'ordre, mieux vaut calculer les puissances "bêtement"
   puisqu'on a besoin de toutes les puissances successives. *)

let ordre t =
  let n = Array.length t in
  let id = id n in
  let u = ref (Array.copy t) in
  let i = ref 1 in
  while !u <> id do
    u := composer !u t;
    i := !i + 1
  done;
  !i

(*
   Attention à bien utiliser "<>" (qui est la negation de "=")
   et pas "!=" (qui est la negation de "=="), sinon on compare
   les adresses des tableaux au lieu de leurs contenus et l'on
   n'obtiendra jamais l'égalité.
*)

let _ = assert (ordre [|1; 2; 0; 4; 3|] = 6)

let periode t i =
  let j = ref t.(i) in
  let per = ref 1 in
  while !j <> i do
    j := t.(!j);
    per := !per + 1
  done;
  !per

let _ =
  let t = [|1;2;0;4;3;5|] in
  assert (Array.map (periode t) t = [|3; 3; 3; 2; 2; 1|])

let estDansOrbite t i j =
  let actuel = ref t.(j) in
  while !actuel <> i && !actuel <> j do
    actuel := t.(!actuel)
  done;
  i = !actuel

let _ =
  let t = [|1; 2; 0; 4; 3; 5|] in
  assert (Array.map  (estDansOrbite t 0) t
          = [|true; true; true; false; false; false|])

(* Une permutation est une transposition ssi elle a
   exactement n - 2 points fixes. *)
let estTransposition t =
  let diff = ref 0 in
  for i=0 to Array.length t - 1 do
    if t.(i) <> i then diff := !diff + 1
```

TD06.ml

```
  done;
  !diff = 2

let _ =
  assert (estTransposition [|0; 2; 1|]);
  assert (not (estTransposition [|0; 1; 2|]));
  assert (not (estTransposition [|1; 2; 0|]))

(* Une permutation est un cycle ssi elle a une seule orbite non triviale. *)
let estCycle t =
  let deplace = ref (-1) in
  let nb_depl =  ref 0 in
  for i = 0 to Array.length t - 1 do
    if t.(i) <> i then
      begin
        deplace := i;
        nb_depl := !nb_depl + 1
      end
  done;
  (!nb_depl <> 0) && (periode t !deplace = !nb_depl)

let _ =
  assert (not (estCycle (id 5)));
  assert (estCycle [|1; 2; 0|]);
  assert (estCycle [|0; 1; 2; 5; 4; 3; 6|]);
  assert (not (estCycle [|1; 0; 2; 4; 3|]))
```