Option informatique : Corrigé du Devoir Surveillé n° 1

Exercice 1

(a)

- `[| 1.0, 2; 2e8, (-42) |]`
- `fun i v -> snd v.(i)`
- `fun f g x -> g (f x)`

(b)

- `int`
- `(int * string) list list`
- `(int * bool * float) option list`

(c)

- `13`
- `[false; true; false]`
- `[12; 20]`

Exercice 2

(a)

```
let rec pgcd a b = match b with
  | 0 -> a
  | _ when b>a -> pgcd b a
  | _ -> pgcd b (a mod b) ;;
```

Cette fonction termine car son second argument décroît strictement à chaque appel, et reste supérieur à 0.

(b)

```
let pgcd2 a0 b0 =
  let a = ref (max a0 b0)
  and b = ref (min a0 b0) in
  while !b > 0 do
    let r = !a mod !b in
    a := !b;
    b := r
  done;
  !a ;;
```

La boucle termine pour la même raison : `!b` décroît strictement à chaque itération.

Exercice 3

(a)

```
let est_premier n =
  let borne = int_of_float (ceil (sqrt (float_of_int n))) in
  let i = ref 2 and result = ref true in
  while !result && !i <= borne && !i < n do
    result := !result && (n mod !i <> 0);
    incr i;
  done;
  !result ;;
```

(b)

```
let cocher i v =
  (* ne pas cocher la case i, car i est premier ! *)
  for j = 2 to (vect_length v - 1) / i do
    v.(i * j) <- false;
  done;;

let crible n =
  let v = make_vect (n+1) true in
  for i = 2 to int_of_float (ceil (sqrt (float_of_int n))) do
    if v.(i) then cocher i v;
  done;
  v;;

let est_premier_crible n =
  let v = crible n in
  v.(n);;

(* ajoute dans "acc" les entiers j tels quel v.(j) vaut "true",
  avec j >= i *)
let rec collecter_indices acc v i =
  if i = vect_length v then acc
  else
    let acc' = if v.(i) then i::acc else acc in
    collecter_indices acc' v (i+1);;

let premiers_inferieur n =
  let v = crible n in
  collecter_indices [] v 2;;
```

Exercice 4

```
let rec swap_list = function
```

```
  | [] -> []
  | (a,b) :: l' -> (b,a) :: swap_list l';;

let rec map_opt f l = match l with
  | [] -> []
  | x :: l' ->
    match f x with
    | None -> map_opt f l'
    | Some y -> y :: map_opt f l';;

let mapi f l =
  let rec mapi_aux f i l = match l with
    | [] -> []
    | x :: l' -> (f i x) :: mapi_aux f (i+1) l'
  in mapi_aux f 0 l;;

let rec grouper l = match l with
  | [] -> []
  | [x] -> [[x]]
  | x::l' ->
    match grouper l' with
    | (y :: first) :: rest when x = y ->
      (x::y::first) :: rest
    | rest -> [x] :: rest;;

let rec grouper_general p l = match l with
  | [] -> []
  | [x] -> [[x]]
  | x::l' ->
    match grouper l' with
    | (y :: first) :: rest when p x y ->
      (x::y::first) :: rest
    | rest -> [x] :: rest;;
```

Exercice 5

```
let rec add x l = match l with
  | [] -> [x]
  | y::_ when x = y -> l
  | y::_ when x < y -> x :: l
  | y::l' -> y :: add x l';;

let rec remove x l = match l with
  | [] -> []
  | y::l' when x = y -> l'
  | y::_ when x > y -> l
```

```
  | y::l' -> y :: remove x l';;

let rec union l1 l2 = match l1 with
  | [] -> l2
  | x::l1' -> add x (union l1' l2);;

let rec intersection l1 l2 = match l1, l2 with
  | [], _
  | _, [] -> []
  | x::l1', y::l2' ->
    if x < y
      then intersection l1' l2
    else if x > y
      then intersection l1 l2'
    else x :: intersection l1' l2';;

(* termine car n decroit a chaque appel *)
let rec parties n p = match p with
  | 0 -> [[]]
  | _ when p > n -> []
  | _ ->
    let avec_n = map (add n) (parties (n-1) (p-1)) in
    let sans_n = parties (n-1) p in
    avec_n @ sans_n ;;
```