

Types, Listes

September 26, 2021

1 Types personnalisés immuables

Une grande force d’Ocaml est de permettre de définir ses propres types. Pour cela, on a deux types de définitions.

Pour le moment, on ne s’intéresse qu’à des type **immuables**, c’est-à-dire dont les valeurs ne peuvent changer.

1.1 Types sommes

1.1.1 Constructeurs constants

Un type somme correspond à une définition de type par cas. Commençons par un exemple très simple, les couleurs pour un jeu de carte :

```
type couleur = Coeur | Pique | Carreau | Trefle
```

On définit ainsi quatre valeurs constantes *Coeur*, *Pique*, *Carreau* et *Trefle* du type *couleur*. On dira que ces noms qui commencent **forcément** par une majuscule sont des **constructeurs** du type *couleur*.

Imaginons qu’on souhaite définir une fonction `numero : couleur -> int` qui renvoie le numéro de la couleur avec la correspondance : *Coeur* = 1, *Pique* = 2, *Carreau* = 3 et *Trefle* = 4. On pourrait le faire avec des `if` :

```
let numero c =  
  if c = Coeur  
  then 1  
  else if c = Pique  
  then 2  
  else if c = Carreau  
  then 3  
  else 4
```

Mais c’est beaucoup plus naturel d’utiliser un filtrage :

```
let numero c =  
  match c with  
  | Coeur -> 1  
  | Pique -> 2  
  | Carreau -> 3  
  | Trefle -> 4
```

1.1.2 Constructeurs paramétrés

On a vu comment définir des constructeurs constants, mais l'intérêt des types sommes est qu'on peut définir des constructeurs paramétrés par une valeur d'un type. Il suffit d'écrire dans un des cas `Constructeur of t` pour que la valeur dépende d'un paramètre, ici de type `t`. Si `x : t`, on pourra alors utiliser le constructeur en écrivant `Constructeur x` comme une application de fonction.

Voici comment on peut maintenant définir la valeur d'une carte :

```
type valeur = Roi | Dame | Valet | Numerale of int
```

Ici on peut donc construire un élément du type `valeur` soit avec les constructeurs constants comme `Dame` soit à l'aide du constructeur paramétré `Numerale 4`.

Quand on effectue un filtrage sur une telle valeur, il suffit alors de considérer le paramètre dans le motif :

```
let point v =  
  match v with  
  | Roi -> 10  
  | Dame -> 10  
  | Valet -> 10  
  | Numerale x -> x
```

On aurait pu aussi filtrer directement sur une valeur comme `| Numerale 3 ->`.

Pour paramétrer avec plusieurs valeurs : on va utiliser un type produit à la place de `t`. Ainsi, on pourrait définir

```
type carte = Joker | Carte of couleur * valeur
```

Attention, maintenant des parenthèses sont nécessaires : `Carte(Carreau, Roi)` y compris dans les filtrages :

```
match crte with  
| Carte(col, v) -> ...
```

1.1.3 Types rékursifs

Là où on comprend toute la richesse de ces types, c'est quand on utilise la récursivité : en effet, un type peut figurer dans sa définition. L'exemple le plus emblématique est celui des listes. Une liste c'est soit vide, soit un maillon contenant un élément puis le reste de la liste. Ainsi, on pourra définir le type des listes d'entiers ainsi :

```
[4]: type liste_entiers = Vide | Cons of int * liste_entiers
```

```
[4]: type liste_entiers = Vide | Cons of int * liste_entiers
```

La liste qui contient 1 puis 2 puis 3 correspond alors à la valeur

```
Cons(1, Cons(2, Cons(3, Vide)))
```

Pour écrire une fonction on peut faire un filtra comme précédemment :

```
let est_vide l =
  match l with
  | Vide -> true
  | Cons(x, q) -> false
```

mais comme on va souvent utiliser la valeur correspondant au type, presque toutes les fonctions seront récursives. Pour calculer la longueur de la liste, on pourra alors faire :

```
let rec longueur l =
  match l with
  | Vide -> 0
  | Cons(x, q) -> 1 + longueur q
```

```
[5]: let rec somme l =
  (* Renvoie la somme des entiers dans l *)
  (* BEGIN SOLUTION *)
  match l with
  | Vide -> 0
  | Cons(x, q) -> x + somme q
  (* END SOLUTION *)
```

```
[5]: val somme : liste_entiers -> int = <fun>
```

```
[6]: assert (somme Vide = 0);
assert (somme (Cons(1,Vide)) = 1);
assert (somme (Cons(1, Cons(2, Vide))) = 3)
```

```
[6]: - : unit = ()
```

Pour la question suivante, comme pour d'autres questions à venir, il est nécessaire de pouvoir produire une erreur quand on ne peut pas calculer quelque chose. On utilise pour cela un mécanisme en OCaml appelé *exceptions*. Pour le moment, on ne rentre pas dans les détails, mais on retiendra juste que pour pouvoir faire une erreur il suffit d'écrire

```
failwith "message a afficher"
```

Une fonction comme

```
let f x = failwith "erreur"
```

a le type 'a -> 'b mais c'est illusion car on ne pourra jamais calculer de valeur avec vu qu'elle produit une erreur de manière inconditionnelle.

```
[7]: let rec nieme l n =
  (* Renvoie le n element de la liste en partant de 0 pour le premier element.
  Produit une erreur si la liste comporte moins de n+1 element *)
  (* BEGIN SOLUTION *)
  match l with
  | Vide -> failwith "Erreur"
```

```
| Cons(x, q) -> if n = 0 then x else nieme q (n-1)
(* END SOLUTION *)
```

```
[7]: val nieme : liste_entiers -> int -> int = <fun>
```

```
[8]: let l = Cons(1, Cons(2, Cons(5, Vide))) in
assert (nieme l 0 = 1);
assert (nieme l 1 = 2);
assert (nieme l 2 = 5);
assert(try let _ = nieme l 4 in false with _ -> true)
```

```
[8]: - : unit = ()
```

1.1.4 Types paramétrés

Il est possible de paramétrer un type par un autre qui ne sera connu qu'au moment de l'utilisation. Reprenons l'exemple précédent, il permet de définir des listes d'entiers, mais si on voulait définir des listes de flottants, il faudrait définir un nouveau type avec d'autres constructeurs. Pourtant, les fonctions comme `somme` ou `nieme` seront presque les mêmes.

Il est possible en OCaml de définir un type de liste générique indépendant du type des éléments. Pour cela, on va réutiliser les types générique déjà vu 'a, 'b, ... :

On ne va pas définir le type `t` mais le type `'a t` pour marquer ce paramètre. Si on a plusieurs paramètres, on va les indiquer entre parenthèses : `('a, 'b) t`.

```
[9]: type 'a liste = Vide | Cons of 'a * 'a liste
```

```
[9]: type 'a liste = Vide | Cons of 'a * 'a liste
```

Ce qui peut sembler magique à première vue c'est qu'il ne va jamais être nécessaire de préciser ce type, il va être automatiquement inféré par OCaml. Par exemple, si on réévalue les fonctions précédentes, on va voir que les types deviennent :

```
val longueur : 'a liste -> int
val somme : int liste -> int
```

car dans `longueur` on ne fait rien des éléments eux-mêmes, donc en l'absence de contrainte, on garde le type générique. Mais dans `somme` comme on les ajoute avec `+`, c'est forcément des éléments du type `int`.

Quand on définit des valeurs, il y a par contre la contrainte très forte d'imposer que tous les éléments soient d'un même type. On ne peut pas écrire `Cons(3, Cons(true, Vide))`. Le constructeur `Vide` n'ayant aucune référence au type `'a`, il est de type `'a liste`.

1.1.5 Un type essentiel 'a option

Il y a un type paramétré essentiel prédéfini dans OCaml, le type 'a option dont la définition est la suivante :

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

qui permet de manipuler des valeurs optionnelles. Mettons qu'un calcul échoue, on peut renvoyer None et quand il réussit Some valeur.

La fonction nieme pourrait ainsi avoir le type nieme : 'a liste -> int -> 'a option en renvoyant None si l'indice donnée est trop grand.

```
[10]: let rec nieme (l:'a liste) (n:int) : 'a option =  
      (* Réécrire la fonction nieme ainsi *)  
      (* BEGIN SOLUTION *)  
      match l with  
      | Vide -> None  
      | Cons(t,q) -> if n = 0 then Some t else nieme q (n-1)  
      (* END SOLUTION *)
```

```
[10]: val nieme : 'a liste -> int -> 'a option = <fun>
```

```
[15]: let l = Cons(1, Cons(2, Cons(5, Vide))) in  
      assert (nieme l 0 = Some 1);  
      assert (nieme l 1 = Some 2);  
      assert (nieme l 2 = Some 5);  
      assert (nieme l 4 = None)
```

```
[15]: - : unit = ()
```

1.1.6 En résumé

On définit un type somme ainsi :

```
type param montype = Constructeur of t | Constante | ...
```

Les paramètres param sont optionnels et le type param montype (ou montype sans paramètres) peuvent apparaître dans les type t des constructeurs.

1.2 Types synonymes

Si on peut définir des listes dont le type des éléments est paramétriques, il est parfois plus lisible d'imposer de considérer, par exemple, des liste d'entiers. Pour cela, on peut définir des types synonymes en OCaml et écrire tout simplement :

```
[11]: type liste_entiers = int liste
```

```
[11]: type liste_entiers = int liste
```

Cela ne sera pas détecté par OCaml quand on évalue une valeur :

```
[12]: Cons(3, Vide)
```

```
[12]: - : int liste = Cons (3, Vide)
```

Cependant, on peut forcer la vérification du type d'une expression en écrivant $(e : t)$ qui signifie à la fois *vérifie que e est du type t* et *affiche e comme étant du type t* :

```
[13]: (Cons(3, Vide) : liste_entiers)
```

```
[13]: - : liste_entiers = Cons (3, Vide)
```

On peut ainsi forcer le type d'une fonction et par exemple faire en sorte que `longueur` ait le type `liste_entiers -> int` :

```
let rec longueur (l : liste_entiers) : int =  
  match l with  
  | Vide -> 0  
  | Cons(x, q) -> 1 + longueur q
```

C'est en général une bonne pratique quand on veut pouvoir réutiliser les fonctions car c'est une manière simple de voir comment on est censé l'appeler et le type de valeur renvoyé.

1.3 Types enregistrements

Les types enregistrements (*record* en anglais) correspondent d'une certaine manière à des tuples dont on aurait nommé les composantes. Ainsi, on peut définir un type `eleve` :

```
[15]: type eleve = {  
  nom : string;  
  age : int;  
  moyenne : float  
}
```

```
[15]: type eleve = { nom : string; age : int; moyenne : float; }
```

Pour créer une telle valeur on utilise `{ nom="Test"; age=4; moyenne=12.5 }` et on peut accéder aux valeurs avec un `.` qui rappelle les attributs en Python :

```
[16]: let e = { nom="Test"; age=4; moyenne=12.5 } in  
e.moyenne
```

```
[16]: - : float = 12.5
```

Pour travailler avec ces éléments, on peut effectuer un filtrage directement :

```
match e with  
{ nom=n; age=a; moyenne=m } -> m
```

mais c'est peu pratique en général par rapport à l'utilisation directe de `e.moyenne`.

```
[17]: let e = { nom="Test"; age=4; moyenne=12.5 } in  
      match e with  
      { nom=n; age=a; moyenne=m } -> m
```

```
[17]: - : float = 12.5
```

2 Les listes en OCaml

En fait, le type des listes est tellement important qu'il est défini directement dans la syntaxe de OCaml. Ainsi on a un type `'a list` avec pour liste vide `[]` à la place de `Vide` et `t::q` à la place de `Cons(t,q)`.

On pourra donc écrire `1::2::3::[]` mais comme c'est assez courant de définir des valeurs ainsi, on peut utiliser directement le raccourci `[1;2;3]` (**Attention** aux `;` et non des `,` comme en Python. Si on se trompe, on va implicitement créer des listes de tuple).

La fonction longueur pourra alors s'écrire

```
let rec longueur l =  
  match l with  
  | [] -> 0  
  | t::q -> 1 + longueur q
```

2.1 Premières fonctions

```
[1]: let tete (l : 'a list) : 'a =  
      (* Renvoie le premier élément de la liste l ou produit une erreur avec  
      failwith "Liste vide" si elle vide *)  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> failwith "Liste vide"  
      | t::q -> t  
      (* END SOLUTION *)
```

```
[1]: val tete : 'a list -> 'a = <fun>
```

```
[2]: let queue (l : 'a list) : 'a list =
  (* Renvoie la queue de la liste l, c'est à dire la liste q telle que l=t::q
  si jamais l est vide produit la même erreur que tete *)
  (* BEGIN SOLUTION *)
  match l with
  | [] -> failwith "Liste vide"
  | t::q -> q
  (* END SOLUTION *)
```

```
[2]: val queue : 'a list -> 'a list = <fun>
```

```
[5]: let l = [1;2;3] in assert(tete l :: queue l = l);
assert(try let _ = tete [] in false with _ -> true);
assert(try let _ = queue [] in false with _ -> true)
```

```
[5]: - : unit = ()
```

```
[9]: let rec somme (l : int list) : int =
  (* renvoie la somme des éléments de la liste d'entiers l *)
  (* BEGIN SOLUTION *)
  match l with
  | [] -> 0
  | t::q -> t + somme q
  (* END SOLUTION *)
```

```
[9]: val somme : int list -> int = <fun>
```

```
[10]: assert(somme [] = 0);
assert(somme [1] = 1);
assert(somme [1;2;3;4] = 10)
```

```
[10]: - : unit = ()
```

```
[11]: let rec nieme (l : 'a list) (n : int) : 'a =
  (* renvoie le n-ieme élément de l en partant de 0
  produit une erreur si l contient moins de n+1 éléments (n'utilisez pas
  ↪ longueur !) *)
  (* BEGIN SOLUTION *)
  match l with
  | [] -> failwith "Indice trop grand"
  | t::q -> if n = 0 then t else nieme q (n-1)
  (* END SOLUTION *)
```

```
[11]: val nieme : 'a list -> int -> 'a = <fun>
```

```
[12]: assert(nieme [1;2;5] 0 = 1);  
assert(nieme [1;2;5] 1 = 2);  
assert(nieme [1;2;5] 2 = 5);  
assert(try let _ = nieme [1;2;5] 3 in false with _ -> true)
```

```
[12]: - : unit = ()
```

```
[34]: let rec contient (x:'a) (l:'a list) : bool =  
    (* renvoie true si et seulement si x est un élément de la liste *)  
    (* BEGIN SOLUTION *)  
    match l with  
    | [] -> false  
    | t::q -> t = x || contient x q  
    (* END SOLUTION *)
```

```
[34]: val contient : 'a -> 'a list -> bool = <fun>
```

```
[35]: assert(contient 0 [1;2;0;3]);  
assert(not (contient 4 [1;2;3;5]));  
assert(not (contient 0 []));  
assert(contient 0 [0])
```

```
[35]: - : unit = ()
```

```
[2]: let rec range (a:int) (b:int) : int list =  
    (* renvoie la liste des entiers a, a+1, ..., b-1 comme  
    on peut le faire avec Python *)  
    (* BEGIN SOLUTION *)  
    if b <= a  
    then []  
    else a :: range (a+1) b  
    (* END SOLUTION *)
```

```
[2]: val range : int -> int -> int list = <fun>
```

```
[5]: assert(range 0 4 = [0;1;2;3]);  
assert(range 1 0 = []);  
assert(range 5 5 = []);  
assert(range (-2) 0 = [-2;-1])
```

```
[5]: - : unit = ()
```

```
[12]: let rec ajoute_fin (x : 'a) (l : 'a list) : 'a list =  
      (* Renvoie la liste obtenue en rajoutant x à la fin de la liste l *)  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> [x]  
      | t::q -> t::(ajoute_fin x q)  
      (* END SOLUTION *)
```

```
[12]: val ajoute_fin : 'a -> 'a list -> 'a list = <fun>
```

```
[7]: assert(ajoute_fin 1 [] = [1]);  
      assert(ajoute_fin 2 [1;3;5] = [1;3;5;2])
```

```
[7]: - : unit = ()
```

```
[14]: let rec concat (l1 : 'a list) (l2 : 'a list) : 'a list =  
      (* Renvoie la liste obtenue en collant l2 à la fin de l1. Voir exemples *)  
      (* BEGIN SOLUTION *)  
      match l1 with  
      | [] -> l2  
      | t::q -> t::(concat q l2)  
      (* END SOLUTION *)
```

```
[14]: val concat : 'a list -> 'a list -> 'a list = <fun>
```

```
[15]: assert(concat [] [1;2] = [1;2]);  
      assert(concat [1;2] [] = [1;2]);  
      assert(concat [1;2] [3;4] = [1;2;3;4])
```

```
[15]: - : unit = ()
```

2.2 Manipulations fonctionnelles des listes

On fait référence ici à des fonctions génériques sur les listes utilisant des fonctions comme argument.

```
[13]: let rec applique (f : 'a -> 'b) (l : 'a list) : 'b list =  
      (* applique la fonction f sur chaque élément de la liste l.  
      si l = [a1; a2; ...; an] alors applique f l renvoie  
      [f a1; f a2; ...; f an] *)  
      (* BEGIN SOLUTION *)
```

```

match l with
| [] -> []
| t::q -> f t :: applique f q
(* END SOLUTION *)

```

[13]: val applique : ('a -> 'b) -> 'a list -> 'b list = <fun>

```

[38]: assert(applique (fun n -> n+1) [1;2;3] = [2;3;4]);
assert(applique (fun x -> x) [] = []);
assert(applique int_of_string ["1";"2";"3"] = [1;2;3])

```

[38]: - : unit = ()

```

[2]: let rec existe (f : 'a -> bool) (l : 'a list) : bool =
    (* renvoie true si et seulement si un élément x de l est tel que f x = true ↵
    ↪ *)
    (* BEGIN SOLUTION *)
    match l with
    | [] -> false
    | t::q -> f t || existe f q
    (* END SOLUTION *)

```

[2]: val existe : ('a -> bool) -> 'a list -> bool = <fun>

```

[3]: assert(existe (fun n -> n mod 2 = 0) [1;3;2;5]);
assert(not (existe (fun n -> n mod 2 = 0) [1;3;5]));
assert(not (existe (fun x -> x) []))

```

[3]: - : unit = ()

```

[4]: (* Écrire une fonction forall sur le modele de existe qui renvoie true si et ↵
    ↪ seulement si
    TOUS les éléments de l vérifient f x = true.
    On pourra utiliser la fonction précédente ou en écrire une ex nihilo. *)
    (* BEGIN SOLUTION *)
    let forall f l = not (existe (fun x -> not (f x)) l)
    (* END SOLUTION *)

```

[4]: val forall : ('a -> bool) -> 'a list -> bool = <fun>

```

[48]: assert(forall (fun n -> n mod 2 = 0) [2;4;6;10]);
assert(not (forall (fun n -> n mod 2 = 0) [2;4;6;11;10]));

```

```
assert(forall (fun x -> x) [])
```

```
[48]: - : unit = ()
```

```
[12]: let rec filtre_options (l : 'a option list) : 'a list =  
      (* Prend une liste l de 'a option et renvoie la liste obtenue en supprimant  
      ↪ tous les None  
      et en renvoyant pour chaque Some x la valeur x. Voir les exemples  
      ↪ ci-dessous *)  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> []  
      | None::q -> filtre_options q  
      | Some x::q -> x :: filtre_options q  
      (* END SOLUTION *)
```

```
[12]: val filtre_options : 'a option list -> 'a list = <fun>
```

```
[53]: assert(filtre_options [None; None; None] = []);  
      assert(filtre_options [Some 3; None; Some 5] = [3; 5])
```

```
[53]: - : unit = ()
```

Une opération importante, complémentaire de `applique` qui applique une fonction sur chaque élément, est `fold` qui effectue un calcul à partir de chaque élément.

Si $l = [a_1; a_2; \dots; a_n]$ alors un `fold` à gauche avec la fonction f et la valeur initiale b va calculer

$$f(\dots f(f(b, a_1), a_2), \dots), a_n)$$

et un `fold` à droite

$$f(a_1, f(a_2, \dots f(a_{n-1}, f(a_n, b)) \dots))$$

```
[1]: let rec fold_gauche (f : 'a -> 'b -> 'a) (b : 'a) (l : 'b list) : 'a =  
      (* Calcule le fold à gauche de f sur l avec la valeur initiale b *)  
      match l with  
      | [] -> b  
      | t::q -> fold_gauche f (f b t) q
```

```
[1]: val fold_gauche : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

```
[2]: let l = [1;2;3;4] in
      assert(fold_gauche (+) 0 l = 10);
      assert(fold_gauche (max) 0 l = 4);
      assert(fold_gauche (min) 10 l = 1);
      assert(fold_gauche (fun s i -> s ^ string_of_int i) "" l = "1234");
      assert(fold_gauche (fun a b -> a) 42 [] = 42)
```

```
[2]: - : unit = ()
```

2.3 Le module List

En fait, la plupart des fonctions qu'on vient de définir existe déjà dans la bibliothèque standard au sein du **module** `List`. Un module est un regroupement de définitions de types et de fonctions qui peut ensuite être appelés directement (sous réserve d'avoir chargé le binaire correspondant, mais c'est le cas pour la bibliothèque standard).

Ainsi il y a dans le module `List` une fonction `mem` qui est équivalente à `contient`. Pour l'appeler sur une liste `l` et un élément `x` on écrit `List.mem x l`.

On peut trouver l'intégralité des fonctions sur la page <https://ocaml.org/api/List.html>. Voici une correspondance avec les fonctions qu'on a défini au dessus :

- `nieme` est `List.nth`, avec une version renvoyant une option `List.nth_opt`
- `tete` est `List.hd` et `queue` est `List.tl`
- `append` est `List.append`, auquel on peut aussi accéder avec `@`. Ainsi `l1@l2` appelle `List.append l1 l2`
- `existe` est `List.exists` et `pourtout` est `List.forall`
- `applique` est `List.map`
- `fold_gauche` est `List.fold_left`, il existe aussi `List.fold_right`

3 Retour sur les modèles de calcul

On va implémenter certains modèles de calcul vus en classe.

3.1 FRACTRAN

On va représenter les fractions $\frac{p}{q}$ par des couples `(p,q)` d'entiers. Un programme est alors une liste de couple (un élément du type `(int * int)` `list`).

```
[8]: type fraction = int * int
      type programme = fraction list

      let p = [ (17,91); (78,85); (19,51); (23,38);
                (29,33); (77,29); (95,23); (77,19);
                (1,17); (11,13); (13,11); (15,2); (1,7); (55,1) ]
```

```
[8]: type fraction = int * int
```

```
[8]: type programme = fraction list
```

```
[8]: val p : (int * int) list =  
      [(17, 91); (78, 85); (19, 51); (23, 38); (29, 33); (77, 29); (95, 23);  
       (77, 19); (1, 17); (11, 13); (13, 11); (15, 2); (1, 7); (55, 1)]
```

```
[9]: let rec premiere_fraction (l : programme) (n : int) : fraction option =  
      (* Renvoie Some(a,b) quand a/b est la première fraction du programme telle  
      ↪ que b divise n.  
      Renvoie None s'il n'y en a pas. *)  
      (* BEGIN SOLUTION *)  
      match l with  
      | [] -> None  
      | (a,b)::q -> if n mod b = 0 then Some (a,b) else premiere_fraction q n  
      (* END SOLUTION *)
```

```
[9]: val premiere_fraction : programme -> int -> fraction option = <fun>
```

```
[26]: assert(premiere_fraction p 2 = Some (15,2));  
      assert(premiere_fraction p 235 = Some (55, 1));  
      assert(premiere_fraction [ (3,4) ] 5 = None)
```

```
[26]: - : unit = ()
```

```
[10]: let rec execute (l : programme) (n : int) (etapes_max : int) : int list =  
      (* execute le programme l en accumulant tous les entiers produit  
      à chaque étape dans une liste qu'on renverra.  
      Le nombre maximum d'étapes, c'est-à-dire la taille maximale de la liste est  
      ↪ donnée  
      par etapes_max. ATTENTION à le prendre en compte pour ne pas faire de  
      ↪ boucles  
      infinies si le programme ne termine pas *)  
      (* BEGIN SOLUTION *)  
      match etapes_max, premiere_fraction l n with  
      | 0, _ -> []  
      | _, None -> []  
      | _, Some (a,b) -> let n = (a*n)/b in  
                          n :: execute l n (etapes_max - 1)  
      (* END SOLUTION *)
```

```
[10]: val execute : programme -> int -> int -> int list = <fun>
```

```
[32]: assert(execute p 2 100 = [15; 825; 725; 1925; 2275; 425; 390; 330; 290; 770;
↪910; 170;
156; 132; 116; 308; 364; 68; 4; 30; 225; 12375; 10875; 28875; 25375; 67375;
↪79625;
14875; 13650; 2550; 2340; 1980; 1740; 4620; 4060; 10780; 12740; 2380; 2184;
↪408;
152; 92; 380; 230; 950; 575; 2375; 9625; 11375; 2125; 1950; 1650; 1450;
3850; 4550; 850; 780; 660; 580; 1540; 1820; 340; 312; 264; 232; 616; 728;
136; 8; 60; 450; 3375; 185625; 163125; 433125; 380625; 1010625; 888125;
2358125; 2786875; 520625; 477750; 89250; 81900; 15300; 14040; 11880; 10440;
27720; 24360; 64680; 56840; 150920; 178360; 33320; 30576; 5712; 2128; 1288;
5320]);
assert(execute [(3,2)] 4 10 = [6; 9])
```

```
[32]: - : unit = ()
```

Comme les variables locales peuvent contenir des fonctions, cela permet de définir des fonctions auxiliaires. Si on se souvient de la fonction `fiboc` du premier TP qui renvoie un couple, on aurait pu définir

```
let fibo n =
  let rec fiboc n =
    if n = 0 then (0,1)
    else let a, b = fiboc (n-1) in
      (b, a+b)
  in
  let a, b = fiboc n in
  a
```

Personne ne peut alors accéder à la fonction `fiboc` en dehors de `fibo`. Cela permet de travailler avec des fonctions ayant parfois plus d'arguments tout en offrant une interface d'appel simple.

```
[18]: let res_de_p n =
  (* renvoie la liste des exposants des puissances de 2 présentes dans
  ↪l'exécution
  de p sur 2 en n étapes maximum.
  On pourra définir une fonction auxiliaire valuation2 : int -> int option
  telle que valuation2 n renvoie None si n n'est pas une puissance de 2
  et Some p si n = 2^p. On pourra ensuite utiliser une fonction définie
  précédemment. *)
  (* BEGIN SOLUTION *)
  let rec valuation2 n =
    if n = 1 then Some 0
    else if n mod 2 <> 0 then None
    else let v = valuation2 (n/2) in
      match v with
      | None -> None
```

```

        | Some p -> Some (p+1)
in let l = execute p 2 n in
let val_l = applique valuation2 l in
filtre_options val_l
(* END SOLUTION *)

```

```
[18]: val res_de_p : int -> int list = <fun>
```

```
[25]: assert(res_de_p 15000 = [2; 3; 5; 7; 11]);
      assert(res_de_p 100 = [2; 3])
```

```
[25]: - : unit = ()
```

3.2 Machines à deux compteurs

On va introduit les types suivants pour représenter les configurations et les programmes d'une machine à deux compteurs. Attention, on commence à numéroté en partant de 0 les instructions.

```
[21]: type configuration = {
      a : int;
      b : int;
      pc : int
    }
type instruction = INCa | INCb | DECa | DECb | JMPa of int * int | JMPb of int *
↳ int
type programme = instruction list

let p = [ JMPa (5, 1); DECa; INCb; INCb; JMPa (0, 0); INCa; DECa ]

```

```
[21]: type configuration = { a : int; b : int; pc : int; }
```

```
[21]: type instruction =
      INCa
    | INCb
    | DECa
    | DECb
    | JMPa of int * int
    | JMPb of int * int
```

```
[21]: type programme = instruction list
```

```
[21]: val p : instruction list =  
      [JMPa (5, 1); DECa; INCb; INCb; JMPa (0, 0); INCa; DECa]
```

```
[23]: let execute_instruction instr conf =  
      (* Execute l'instruction instr sur la configuration conf et renvoie la  
      ↪nouvelle configuration *)  
      (* BEGIN SOLUTION *)  
      match instr with  
      | INCa -> {a=conf.a+1;b=conf.b;pc=conf.pc+1}  
      | INCb -> {a=conf.a;b=conf.b+1;pc=conf.pc+1}  
      | DECa -> {a=conf.a-1;b=conf.b;pc=conf.pc+1}  
      | DECb -> {a=conf.a;b=conf.b-1;pc=conf.pc+1}  
      | JMPa(i, j) -> {a=conf.a;b=conf.b;pc=if conf.a = 0 then i else j}  
      | JMPb(i, j) -> {a=conf.a;b=conf.b;pc=if conf.b = 0 then i else j}  
      (* END SOLUTION *)
```

```
[23]: val execute_instruction : instruction -> configuration -> configuration =  
      <fun>
```

```
[24]: let rec execute prog conf =  
      (* Execute le programme prog sur la configuration conf et  
      renvoie la configuration finale, c'est-à-dire quand conf.pc pointe en  
      ↪dehors du programme.  
      On pourra utiliser List.nth_opt *)  
      (* BEGIN SOLUTION *)  
      match List.nth_opt prog conf.pc with  
      | None -> conf  
      | Some instr -> execute prog (execute_instruction instr conf)  
      (* END SOLUTION *)
```

```
[24]: val execute : instruction list -> configuration -> configuration = <fun>
```

```
[30]: assert(execute p {a=4;b=0;pc=0} = {a=0;b=8;pc=7});  
      assert(execute p {a=2;b=2;pc=0} = {a=0;b=6;pc=7})
```

```
[30]: - : unit = ()
```