C. Charignon

Table des matières

Première partie
Cours

1 Introduction

Jusqu'ici pour enregistrer plusieurs valeurs on a utilisé des tableaux (type « `Array` » en Caml et `List` en Python). Le fonctionnement est simple : pour un tableau de n cases censé contenir des éléments d'un type t, on réserve en mémoire n blocs consécutifs de la taille nécessaire à enregistrer un élément de type t (64 bits pour un nombre par exemple).

Si on connaît l'adresse du début du tableau, alors pour tout $i \in [\![0, n-1]\!]$, le contenu de la i-ème case est situé à l'adresse : début du tableau + $i\times$ taille d'un bloc.

On peut donc accéder directement à la $i$ème case, la lire, la modifier, etc... Et cet accès est en $O(1)$ (il faut faire une multiplication).

Par contre, agrandir le tableau est compliqué : s'il ne reste pas de place après le dernier bloc, il va falloir chercher un autre endroit de la mémoire, plus grand, et y recopier tout le tableau. Soit un coût en $O(n)$.

N.B. Toutes les cases du tableau doivent avoir la même taille. C'est pourquoi Caml impose à tous les objets contenus dans le tableau d'être de même type.

Nous allons maintenant découvrir un nouveau type de donnée permettant au contraire facilement de rajouter un élément. Il s'agit des listes chaînées (ou juste liste), appelé le type « `List` » en Caml.

Le fonctionnement est le suivant : on enregistre le premier élément de la liste et juste à côté l'adresse mémoire du second élément (on dit un "pointeur" vers le second élément). À côté du second élément se trouve l'adresse du troisième, etc...

Pour rajouter un élément x c'est facile : on enregistre x et juste à côté l'adresse de l'ancien premier élément. Rien à modifier dans le reste de la liste, donc l'ajout est en $O(1)$.

Par contre, si on veut accéder au $i$ème élément, il faut partir du début, lire l'adresse du second élément, y aller, lire l'adresse du troisième élément, etc... Ceci fait $O(i)$ opérations.

2 Concrètement en Caml

2.1 Opérations élémentaires sur tableaux et listes

Nous n'utiliserons pas de tableau en Caml avant plusieurs chapitres... Je note ici les commandes principales plus pour comparer les commades utiles pour les tableaux et pour les listes.

- **Opérations élémentaires sur les tableaux) :**
  - ◇ créer un tableau vide de n cases contenant initialement l'élément x : `Array.make n x`
  - ◇ modifier la i case : `monTableau.(i) <- nouvelleValeur`
  - ◇ lire la $i$ème case : `monTableau.(i)`

- **Opérations élémentaires sur les listes :**
  - ◇ La liste vide : `[]`
  - ◇ La liste L à laquelle on rajoute l'élément x : `x::L`. L'opérateur `::` est parfois appelé "cons", c'est le constructeur des listes. Par ailleurs, si $L' = x :: L$, on dit que x est la "tête" de L', et que L en est la "queue". Il existe des fonctions renvoyant la tête et la queue d'une liste : ce sont `List.hd` et `List.tl` ("head" et "tail")

Les quelques autres opérations possibles sont décrite dans le document résumant les commandes Caml. Elles permettent de gagner du temps, mais en fait seules la liste vide `[]` et le constructeur `::` sont vraiment indispensables. *Remarque :* L'opérateur `::` devrait être appelé le « constructeur des listes ». Mais comme le type des listes est le plus fréquemment utilisé dans les langages fonctionels comme Caml, on dit souvent juste « constructeur » ou même « cons ».

Il est important de noter que les opérations ci-dessus s'exécutent en $O(1)$ (c'est-à-dire en temps borné).

Enfin, comme on le remarque sur ces commandes élémentaires, les tableaux sont modifiable ("mutable" en anglais, ou encore impératifs). Alors que les listes sont "immuables", ou encore "persistante". Ainsi, une liste n'est pas modifiable, les opérations créent une nouvelle liste.

Par exemple `x::L` **renvoie** une liste, c'est une expression de type liste.

Alors que `monTableau.(i) <- nouvelleValeur` modifie le tableau, mais ne renvoie rien. C'est une expression de type "unit".

Remarque : L'opération `x::L` renvoie une nouvelle liste, mais les données contenues dans L ne sont pas pour autant recopiées : on crée juste une nouvelle case qui pointe vers l'ancienne liste. L'opération se fait en $O(1)$.

2.2 Définition formelle

Définition 2.1. *Une liste peut être :*

- *la liste vide (*`[]`*) ;*
- *ou un élément ajouté devant une liste (*`(un élément) :: (une liste)`*).*

Cette définition est récursive ! Elle est particulièrement adapté à la programmation : nos programmes vont tout simplement suivre cette définition.

2.3 Filtrage

2.3.1 Principe

Le langage Caml fournit un mécanisme extrémement utile pour manipuler les type construits comme les listes : le filtrage par motif, « pattern matching » en anglais. Voici un exemple :

```
let truc l=
    match l with
        | [] -> bla
        | t::q -> blabla  (* Les identifiants t et q tels que l=t::q sont automatiquement créé
    ↪ s
         et utilisables dans ce cas*)
```

Il faut se dire que le premier cas est atteint si `l=[]`. Et le second est atteint si $\exists t, q$ tq $l = t :: q$. Et dans ce cas, le `t` et le `q` qui existent sont automatiquement définis.

2.3.2 Mise en garde

De manière générale, répondre à une question de type $\exists$ est difficile ! Ne croyez pas que Caml peut faire des miracles... En réalité il ne peut effectuer cette opération que dans des cas extrèmement balisés : lorsque le motif utilisé n'utilise que des comstructeurs.

Par exemple :

```
let nimp x=
  match x with
      | a + b -> a
```

Déjà il n'existe pas d'unique couple (a, b) tel que $x = a + b$, de sorte que a et b sont mal définis ici.

```
let nap x=
  match x with
     | a+1 -> a
;;
```

Ici, il existe effectivement un unique a tel que $x = a + 1$. Mais ceci repose sur des propriétés de l'addition non triviales (existence d'un élément symétrique), Caml est incapable de le deviner.

Au final, on ne peut utiliser dans un motif que des « constructeurs », c'est-à-dire les fonctions élémentaires utilisées pour définir un type.

En revanche, la virgule `(,)` est en fait le constructeur des couples (ou des p-uplets plus généralement) ce qui fait qu'on peut l'utiliser dans un motif :

```
let symetrie m=
  match mwith
      | (x,y) -> (y,x)
;;
```

2.3.3 Deuxième mise en garde et clauses gardées

Deuxième mise en garde : ne pas utiliser un identifiant déjà existant dans un motif. Par exemple

```
let rec appartient x l=
    match l with
    |x::q -> true
    |t::q -> appartient x q
```

ne fonctionne pas. Lors de l'utilisation du motif de filtrage `x::q`, un *nouvel* identifiant x sera créé, mais Caml n'ira pas tester si ce nouveau x est égal au x de l'argument de la fonction. L'argument x est écrasé...

En conclusion : ne jamais utiliser un identifiant déjà lié dans un motif de filtrage.

Voici comment corriger l'exemple précédent : on dispose d'une commande `when` pour rajouter des conditions lors d'un filtrage :

```
let rec cherche x l=
    match l with
    |t::q when t=x  -> true
    |t::q     -> cherche x q
```

On note que le t est inutile dans le dernier cas, autant écrire :

```
let rec cherche x l=
    match l with
    |t::q when t=x  -> true
    |_::q     -> cherche x q
```

pour économiser un peu de mémoire.

Remarque : Dans la fonction ci-dessus, rien ne permet de connaître le type des éléments de l. Par conséquent, elle est utilisable avec n'importe quel type.

2.3.4 Bonus : quelques syntaxes pour gagner un peu de temps

- le souligné « _ » : Si on a pas besoin d'une valeur dans un motif, on peut utiliser le souligné pour éviter d'avoir à lui attribuer un identifiant. Exemple : Si par exemple on n'a pas besoin de la valeur de t on peut utiliser _ à la place :

  ```
  let longueur l=
      match l with
          | [] -> 0
          | _::q -> 1 + longueur q (* seul q est créé*)
  ```

- la syntaxe `function` : Pour écrire une fonction qui va immédiatement filtrer l'argument qui lui est fourni, il existe une syntaxe raccourcie. Sur l'exempli précédent :

  ```
  let longueur = function
          | [] -> 0
          | _::q -> 1 + longueur q
  ```

  Remarquez qu'on n'écrit pas le « `match ... with` » mais également qu'on ne donne pas de nom à l'argument. Dans certains cas, cela peut être un soulagement de ne pas avoir à trouver un n-ième identifiant pour une valeur qui n'interviendrait que deux fois dans le code.

3 Bonus : récursivité terminale

Considérons les deux exemples suivant pour calculer le nombre d'occurrence d'un élément dans une liste :

```
let rec nbOcc1 x l=
   match l with
       |[]  ->0
       |t::q when t=x  -> 1+ nbOcc1 x l
;;
```

```
let rec nbOcc2Aux accu x l=
   match l with
       |[]  -> accu
       |t::q when t=x  -> nbOcc2Aux (accu+1) x l
;;

let nbOcc2 = nbOcc2Aux 0;;
```

Le deuxième version utilise un argument supplémentaire qui sert d'accumulateur.

Remarque : Quand vous créez une fonction auxiliaire, et/ou utilisez un argument supplémentaire non mentionné dans l'énoncé, il faut expliquer son rôle. Ici la description de `nbOcc2Aux` serait : « Renvoie accu + (le nombre de x dans l) ».

De sorte que `nbOcc2Aux 0 x l` renvoie 0 + le nombre de x dans l, qui est bien le résultat cherché.

En gros, on est tenta d'utiliser un accumulateur à chaque fois qu'on utiliserait une variable en programmation impérative. *Mais on peut s'en passer lorsque cette variable est justement le résultat qu'on veut renvoyer.*

Maintenant, hormis les complications pour écrire le programme, qu'est-ce que cela change pour la machine ?

Et bien, la version avec accumulateur est légèrement optimisée. En effet, la pile des appels sera plus petite, elle va même toujours rester à un seul élément.

C'est du au fait que la fonction renvoie directement `nbOcc2Aux (accu+1) x q` sans autre opération derrière.

Une telle fonction est dite récursive terminale.

Remarque : Python ne détecte pas la récursivité terminale, et n'optimise donc pas la pile des appels... [1]

Deuxième partie

Exercices

1. It's not a bug, it's a feature.

Exercices : listes

Pour ce TP comme pour les suivants, créez un fichier `listes.ml` *dans lequel vous mettrez toutes les fonctions sur les listes qui n'existent pas déjà dans OCaml et dont vous pensez qu'elles pourront vous servir un jour. En particulier, mettez-y les fonctions venant d'exercices signalés par !!.*

Exercice 1. * Représentation mémoire

Le tableau ci-dessous symbolise (de manière très simplifiée) l'état de la mémoire à un instant donné. On a représenté 50 blocs mémoire, chacun contient soit une valeur, soit un couple (valeur |# pointeur vers un autre bloc).

2.7	#31	38	#22	12	#38	−5.0	#29	15		1.3	#08	0.4	#13	39	#09	2	#21	−5	#12
01		02		03		04		05		06		07		08		09		10	

−3		8	#19	2.4	#37	−6		1		[]		6.1		1.3		2	#16	−3	
11		12		13		14		15		16		17		18		19		20	

9	#33	1	#45	−11	#25	1.2		3	#27	3.6		9	#10	56		−1.	#07	−7	
21		22		23		24		25		26		27		28		29		30	

2.9	#34	13		0	#03	[]		[]		1.3		0.4	#01	39	#35	2		−5	
31		32		33		34		35		36		37		38		39		40	

3.1		0	#23	−0.3		0		7	#42										
41		42		43		44		45		46		47		48		49		50	

1. On suppose que l'identificateur `liste1` contient le pointeur `#02`.
   Quelle est la liste correspondant à `liste1` ? Est-ce une liste d'entiers ou de réels ?
2. On exécute l'instruction Caml suivante : `let liste2 = (List.tl liste1) ;;`.
   Quel est le pointeur que contient l'identificateur `liste2` ?
3. On exécute l'instruction Caml suivante : `let liste3 = 10::liste1 ;;`
   En supposant que Caml utilise, s'il en a besoin, la première adresse mémoire libre, quel est l'état de la mémoire de Caml à l'issue de cette instruction ?

Exercice 2. *! Fonctions élémentaires sur les listes

Donner les fonctions suivantes :

1. Calcul de la longueur d'une liste (existe déjà sous le nom de `List.length`)
2. Calcul de la somme des éléments d'une listes d'entiers
3. Calcul du maximum d'une liste
4. Tester si un élément est dans une liste (existe déjà dans Caml sous le nom de `mem` comme « member »)
5. Tester si deux listes sont égales (existe déjà : opérateur `=`)
6. Concaténer deux listes (existe déjà, c'est l'opérateur `@`). Préciser la complexité de cette fonction.

Exercice 3. *!! Intervalle

Écrire une fonction `intervalle` prenant en argument deux entiers a et b et renvoyant la liste $[\![a, b[\![$ (dans l'ordre croissant).
Remarque : En Python, on obtient le tableau contenant les entiers de $[\![a, b[\![$ via `list(range(a,b))`.

Exercice 4. **! Retourner une liste

1. Écrire une fonction prenant une liste `l` et renvoyant la liste contenant les mêmes éléments que `l` mais dans l'ordre inverse.
   Remarque : En OCaml c'est `List.rev`.
2. Calculer la complexité de votre fonction.
3. Si le retournement d'une liste de longueur n n'est pas $O(n^2)$, améliorez-la et retraitez cette question et la précédente. [2]

2. Ces deux questions sont donc récursives.

Exercice 5. **! Fonctions (à peine) plus difficiles

Donner la complexité de vos fonctions. On comptera le nombre d'utilisation du constructeur `::`, à la lecture (dans le filtrage), comme à l'écriture (construction de la liste à renvoyer).

1. Enlever tous les doublons d'une liste
2. Enlever toutes les occurrences d'un élément dans une liste. Puis la première occurrence d'un élément dans une liste. Et enfin la dernière.
3. Déterminer l'indice d'un élément x dans une liste l. On renverra une erreur si x n'est pas dans l.

Exercice 6. ** Un peu d'arithmétique

1. Écrire une fonction récursive renvoyant le plus petit diviseur $\geqslant 2$ d'un entier.
2. Soit $n \in \mathbb{N}$ et d son plus petit diviseur $\geqslant 2$. Montrer que d est premier.
3. Écrire une fonction récursive prenant un entier et renvoyant la liste de ses facteurs premiers
4. Écrire une fonction récursive prenant un entier n et renvoyant la liste des chiffres pour écrire n en base 2. Pour faciliter la programmation, on pourra décider de mettre le chiffre des unités en tête de liste (bits de poids faible d'abord). On rappelle que le reste de la division euclidienne de n par 2 s'obtient par `n mod 2`.

Exercice 7. ** Des chiffres et des lettres, simplifié

Le but est d'écrire une fonction `sommeFait` prenant un entier n et une liste d'entiers positifs `l` et indiquant si on peut choisir des éléments de `l` dont la somme fais n. Les répétitions sont autorisées. Par exemple `sommeFait 13 [7;5;3]` renverra `true`, tandis que `sommeFait 13 [7;8]` renverra `false`. On convient que la somme d'aucun nombre vaut 0.

1. Soit `l` une liste non vide. Soit `t, q` tels que `l=t::q`. Montrer qu'on peut obtenir n comme somme d'éléments de `l` si et seulement si on peut obtenir n comme somme d'éléments de `q` ou on peut obtenir $n-t$ comme somme d'éléments de `l`.
2. Programmer en Caml une fonction `sommeFait` correspondante.
3. Démontrer que le programme termine et est correct. On pourra utiliser une récurrence forte sur $n+|l|$. ($|l|$ désigne ici la longueur de l.)
4. Variante : écrire une fonction qui renvoie la liste des nombres permettant d'obtenir n si c'est possible, et `[-1]` sinon.

Exercice 8. ** Ordre lexicographique

On définit l'ordre lexicographique sur les listes d'entiers de la manière suivante : On compare les premières composantes des deux listes : la liste dont la première composante est la plus petite est considérée comme inférieure à l'autre. En cas d'égalité des premières composantes, on compare les deuxièmes composantes, et ainsi de suite jusqu'à, s'il le faut, épuisement de l'une des deux listes. Dans ce dernier cas, la liste la plus courte est considérée comme la plus petite. Écrire une fonction Caml nommée `inferieur` qui prend comme arguments deux listes d'entiers et qui renvoie `true` si et seulement si la première liste est inférieure ou égale à la deuxième (pour l'ordre lexicographique).
Remarque : Caml propose en fait déjà cette fonction, c'est l'opérateur `<`. Cet opérateur permet également de comparer entiers, flottants... On dit que c'est un opérateur « polymorphe » car il peut utiliser différents types de données.

Exercice 9. ** Base 2

Les fonctions vues au chapitre sur la représentation des entiers sont souvent plus facilement écrites récursivement avec des listes. Dans cet exercice, un entier positif sera représenté par son écriture en base 2 « à l'envers » : le bit des unités (poids faible) en premier. Ceci permet au passage d'accroître à souhait la taille de la liste : on ne se limitera pas à 64 bits.

1. Écrire une fonction prenant en entrée deux listes d'entiers $l1$ et $l2$ représentant deux nombres écrits en binaire et renvoyant la liste contenant l'écriture en binaire de la somme des deux nombres.
2. Écrire une fonction renvoyant le produit de deux entiers écrits en base 2.
3. Écrire les fonctions de conversion base 2 vers base 10 et réciproque.

Exercice 10. *! Programmation fonctionnelle**

La maîtrise de ce genre de fonctions pourra vous faire gagner beaucoup de temps...

1. On rappelle qu'un prédicat est une fonction renvoyant un booléen.
   (a) Écrire une fonction `existe p l` prenant un prédicat et une liste et cherchant s'il existe un élément de l qui vérifie p. (En OCaml : `List.exists`.)
   (b) De même, écrire une fonction `pourTout p l` vérifiant si le prédicat est réalisé pour tout élément de l. (En OCaml : `List.for_all`)

(c) Enfin, écrire une fonction `existeUnique p l` cherchant s'il existe un unique élément de l vérifiant p.

(d) À l'aide de la fonction `existe` et de la fonction `mem` de Caml, écrire une fonction `intersection_non_vide` qui teste si deux listes ont au moins un élément en commun.

2. Écrire une fonction `appliquee` prenant en argument une fonction f et une liste `[a0; ...; an]`, qui renvoie la liste `[ f a0; ...; f an]`. Quel sera le type de cette fonction ?
   Dans OCaml c'est : `List.map`.
3. *(exemple:)* Comment récupérer la liste des ordonnées d'une liste de couples ?
4. Écrire une fonction `appliqueeDouble` qui prend en entrée deux listes et une lci et qui renvoie la liste obtenue en appliquant la lci aux listes composante par composante.
5. Écrire une fonction `implosion` prenant en entrée une liste l, une loi de combinaison interne $*$ et un élément neutre pour $*$ et qui calcule $_{x\in l}x$.
   Par exemple, si la loi est + on obtiendra la somme, si c'est $\times$ on obtiendra le produit. Chercher d'autres exemples.
   N.B. Dans OCaml : `List.fold_left` et `List.fold_right`, selon que l'on commence les calculs par la droite ou par la gauche de la liste.
6. Réécrire une fonction `pour_tou` en se basant sur `implosion`.
7. En utilisant les questions précédentes, ainsi que la fonction `intervalle` de l'exercice 3, écrire, en une ligne, une fonction pour :
   (a) Calculer la somme des n premiers entiers
   (b) Calculer la somme des n premiers carrés
   (c) Calculer la série harmonique $\sum_{k=1}^{n} \frac{1}{k}$
8. Écrire une fonction `filtre` prenant en entrée un prédicat p et une liste l qui renvoie la liste des éléments de l qui vérifient p.
   N.B. Dans Ocaml : `List.filter`
9. En déduire une fonction prenant une liste et renvoyant la somme des carrés de ses nombres positifs.
10. Écrire en une ligne une fonction calculant l'intersection de deux listes.

Les deux dernières questions utilisent des procédures.

11. Écrire une fonction `execute` qui prend en entrée une liste `l` et une procédure `p` et applique `p` à chaque élément de `l`. Écrire en une ligne une fonction qui affiche le contenu d'une liste d'entiers.
    N.B. C'est `List.iter` en Ocaml.
12. Pouvez-vous obtenir la fonction précédente en vous basant sur `compose` ?

Quelques indications

2 Pour la concaténation : la complexité dépend de la longueur d'une des deux listes seulement.

3 Prendre juste le temps de se demander s'il vaut mieux procéder par récurrence sur a ou b.

4 Pour une version efficace : créer une fonction auxiliaire prenant un argument supplémentaire : les éléments de liste déjà traités.

5
1. Utiliser `List.mem`
2. Pour enlever la dernière occurrence : avec un retournement, on obtient $O(n)$. Mais la manière la plus efficace consiste à écrire une fonction qui pour tout élément x et liste l renvoie le couple (l privé de son dernier x, $x \in l$).

6 1) Écrire une fonction auxiliaire prenant en entrée un argument supplémentaire : le prochain entier à essayer. Ainsi :

```
let rec plusPetitDivAux n i =
   (* Renvoie le plus petit diviseur de n qui soit  i *)
   ...
```

9
1. Utiliser une fonction auxiliaire qui renvoie une valeur supplémentaire : la retenue.
2. Écrire au préalable une fonction pour ajouter i zéros en têt d'une liste.

Quelques solutions

1

2

3

```
let rec intervalle a b=
  (* Renvoie l'intervalle d'entiers [a,b[ *)
  if b > a then
    a::(intervalle (a+1) b)
  else []
;;
```

5

```
let rec sansLes x l=
   (* Renvoie l privé de tous les x qu'il pourrait y avoir *)
   match l with
    |[] -> []
    |t::q -> if t=x then sansLes x q
        else t::sansLes x q
;;

(* Avec une clause gardée : *)
let rec sansLes x l=
   (* Renvoie l privé de tous les x qu'il pourrait y avoir *)
   match l with
    |[] -> []
    |t::q when t=x -> sansLes x q
    |t::q        -> t::sansLes x q
;;

let rec sansLePremier x l=
   (* Renvoie l privé de son premier x *)
   match l with
    |[] -> []
    |t::q when t=x ->  q
    |t::q        ->  t::sansLePremier x q
;;
sansLePremier 2 [5;4;1;2;1;4;2;1;4;2];;

sansLes 2 [5;4;1;2;1;0;2];;

(* Première méthode : quand on rencontre un x, on utilise List.mem pour savoir si c'est le
    ↪ dernier *)
let rec sansLeDernier x = function
   |[] -> []
   |t::q when t=x && not (List.mem x q)->  (* Ici, t est le dernier x de la liste *)
        q
  |t::q  -> t:: sansLeDernier x q
;;


(* Deuxième méthode : retourner la liste *)
(* List.rev *)
let sansLeDernier x l=
    List.rev (sansLePremier x (List.rev l));;
sansLeDernier 2 [5;4;1;2;5;4;1;2;1;2];;


(* Troisième méthode : écrire une fonction qui fonction qui renvoie un couple
     (l privé de son dernier x, x appartient à l)    *)
 (* Cela revient à calculer sansLeDernier et List.mem simultanément *)

```

```
let rec sansLeDernierAux x l=
   match l with
    |[] -> [], false
    |t::q when t=x -> let (q_sans_son_dernier_x, x_dans_q) = sansLeDernierAux x q in
                                  if x_dans_q then t::q_sans_son_dernier_x, true
                                  else q_sans_son_dernier_x, true
                |t::q ->  let (q_sans_son_dernier_x, x_dans_q) = sansLeDernierAux x q in
                          t::q_sans_son_dernier_x,  x_dans_q
;;
let sansLeDernier3 x l=
  fst (sansLeDernierAux x l);;
sansLeDernier3 2 [5;0;5;6;2;5;2;1;2;3];;
```

6

```
let plusPetitDiviseur  n  =
  let rec diviseurAux  n  d  =
    if n mod d = 0 then d
    else diviseurAux n (d+1);
  in
    diviseurAux n 2
;;

let rec facteursPremiers  n  =
  match n with
    |0 -> failwith "0 a une infinité de facteurs premiers"
    |1 -> []
    |(-1) -> []
    |_ -> let d = plusPetitDiviseur n in
      d :: facteursPremiers (n/d)
;;

let rec enBase2  n  b  =
  match n with
    |0 -> []
    |_ -> (n mod b) :: enBase2 (n/b) b
  ;;
```

7 1. Soit `l` une liste et $n \in \mathbb{N}$. Supposons qu'il existe (t, q) tels que `l=t::q`. Pour obtenir n comme somme d'éléments de `l`, il y a uniquement les deux possibilités suivantes :

- Prendre t, puis obtenir $n - t$ comme somme d'éléments de `l` (on peut reprendre t car les répétitions sont autorisées ;
- Ne pas prendre t, auquel cas il faut obtenir n comme somme d'éléments de `q`.

2.

```
let rec sommeFait n l=
  (* Indique si on peut faire n en sommant des éléments de l (répétitions autorisées) *)
  if n<0 then false
  else if n=0 then true
  else
  match l with
  |[] -> n=0
  |t::q -> sommeFait (n-t) l || sommeFait n q
;;
```

3. ⚠ Pour rédiger la démonstration sans devoir utiliser des kilomètres de périphrases, il est utiles de poser quelques notations.

   Notons E l'ensemble des listes d'entiers positifs. Notons pour tout $l \in E$ et tout entier n, $SF(n, l)$ le booléen « On peut obtenir n comme somme d'éléments de `l` ». Notre but est donc de prouver que pour tout $l \in E$ et tout entier n, `sommeFait n l= SF(n,l)`.

   Pour tout $k \in \mathbb{N}$, posons $P(k)$: « Pour toute liste `l` et tout entier n tel que $n + |l| = k$, on a `sommeFait n l= SF(n,l)` ».

   - **Initialisation :** Soit $l \in E$ et $n \in \mathbb{N}$ tels que $|l| + n = 0$. Si $n < 0$, on ne peut pas obtenir n comme somme d'entiers positifs donc $SF(n, l) = \bot$ (faux). C'est aussi le résultat renvoyé par `sommeFait n l`.

     Si $n \geqslant 0$, alors comme $|l| \geqslant 0$, c'est que $n = 0$ et `l=[]`. Alors $SF(n, l) = \top$ (Vrai) car 0 est somme d'aucun nombre. C'est bien le résultat renvoyé par `sommeFait n l`.

     Ainsi, dans tous les cas, `SF(n,l)= sommeFait n l` .

- **Hérédité :** Soit $k \in \mathbb{N}$. Supposons que $\forall l \in [\![0,k]\!]$, $P(l)$. Soit $l \in E$ et $n \in \mathbb{Z}$ tel que $|l|+n = k+1$.
  - ◇ Si $l = []$, alors $n = k+1 > 0$ et il n'est pas possible d'obtenir n comme somme d'éléments de l. Or, `sommeFait n l` renvoie bien $\bot$.
  - ◇ S'il existe t, q tel que $l = t :: q$, alors vu la première question, on a $SF(n,l) = SF(n-t,l) \vee SF(n,q)$ ($\vee$ est le symbole mathématique pour « ou »). Or précisément, `sommeFait n l` renvoie `sommeFait (n-t) l` $\vee$ `sommeFait n q`. De plus, par hypothèse de récurrence, `sommeFait (n-t) l` renvoie $SF(n-t,l)$ et `sommeFait n q` renvoie $SF(n,q)$. On obtient bien que `sommeFait n l = SF(n,l)`.

Ainsi par récurrence, pour tout $n \in \mathbb{N}$ $P(n)$.

4.
```
let rec sommeQuiFait n l =
  (* Entrée : n, entier
              l, liste d'entiers positifs *)
  (* Renvoie une liste d'éléments de l dont la somme vaut n s'il en existe, et [-1] sinon.
    ↪  *)
  if n<0 then [-1]
  else if n=0 then []
  else match l with
       |[] -> [-1]
       |t::q -> match sommeQuiFait (n-t) l with (* Essai en prenant t *)
                | [-1] -> sommeQuiFait n q (* Dernier essai : sans prendre t *)
                | res -> t::res
;;
sommeQuiFait 12 [8;5;2];;
```

8

9

10
```
let rec existe p = function
  |[] -> false
  |t::q when p t -> true
  |t::q -> existe p q
;;

existe (fun x -> x>0) [-9;-2;-3];;

let rec pour_tout p = function
  |[] -> true
  |t::q when p t -> pour_tout p q
  |t::q -> false
;;

pour_tout (fun x -> x>0) [9;2;3];;

let rec existe_unique p=function
  |[] -> false
  |t::q when p t -> not (existe p q)
  |t::q -> existe_unique p q
;;

existe_unique (fun x -> x>0) [-9;-2;-3];;

let intersection_non_vide l1 l2=
  existe
    (fun x-> List.mem x l1)
    l2
;;

intersection_non_vide [2;5;3] [9;5;0];;

let rec appliquee f =function
```

```
  |[] -> []
  |t::q -> f t :: appliquee f q
;;

List.map snd [(1,2); (3,4)];;

let rec appliquee_double lci l1 l2 =
  match l1, l2 with
  |[],[] -> []
  |[],_ -> failwith "pas la même longueur"
  |_, [] -> failwith "pas la même longueur"
  |t1::q1, t2::q2 -> (lci t1 t2) :: appliquee_double lci q1 q2
;;

appliquee_double (+) [1;2;3] [4;5;6];;

List.fold_left;;
List.fold_right;;



let rec implosion lci  l e =
  match l with
  |[] -> e
  |t::q -> lci t (implosion lci  q e)
;;

(* NB : ici les calculent commencent « à droite » de la liste :
  implosion * [x0; ... ; xk] e renvoie   x0 *  ... (x(k-1) * ( xk*e)))
C'est la fonction List.fold_right de Caml *)

(* Pour commencer les calculs par la gauche, on peut utiliser un accu en argument supplé
    ↪ mentaire :
( Le résultat sera le même pour les deux fonction si la lci est associative ) *)

let implosion_gauche lci e l = (* rema : le neutre sera utilisé « à gauche » de la liste *)

  let rec aux accu = function
    |[] -> accu
    |t::q -> aux (lci accu t) q
  in

  aux e l
;;


(*Exemple amusant : redéfinissons pour_tout *)
let pour_tout p l=
  implosion
    (&&)
    (List.map p l)
    true
;;

(* NB : pour utiliser des fonctions ayant de gros argmuments comme ici implosion, je mets
    ↪ souvent un argument par ligne : c'est plus lisible. *)

pour_tout (fun x-> x>0) [4;-5;-6];;

(* Sauriez-vous faire de même pour il_existe ? *)


let rec intervalle a b=
  (* Renvoie l'intervalle semi-ouvert [|a,b[| *)
```

```
  if a>=b then []
  else a::intervalle (a+1) b
;;

let somme_premiers_entiers n=
  implosion
    (+)
    (intervalle 0 n)
    0
;;
somme_premiers_entiers 5;;

let somme_premiers_carres n=
  implosion
    (+)
    (List.map (fun x-> x*x) (intervalle 0 n) )
    0
;;
somme_premiers_carres 4;;

let serie_harmo n=
  (* Attention : on manipule ici des flottants *)
   implosion
    (+.)
    (List.map
       (fun x-> 1./. float_of_int n)
       (intervalle 0 n)
    )
    0.
;;


let rec filtre p = function
  |[] -> []
  |t::q when p t -> t::filtre p q
  |_::q -> filtre p q
;;

let somme_carres_des_positifs l =
  implosion (+)
    (List.map
      (fun x-> x*x)
      (filtre (fun x-> x>0) l)
    )
    0
;;

somme_carres_des_positifs [1;-3;4];;

let intersection l1 l2=
  filtre (fun x-> List.mem x l1) l2;;

intersection [5;6;7] [3;4;5;1];;
```