



INF201
Algorithmique et Programmation Fonctionnelle
Cours 6 : Listes

Année 2018 - 2019

$f(x)$



Programmation fonctionnelle en OCaml :

- ▶ types de base :
booléens, entiers, réels, caractères, chaînes de caractères
- ▶ identificateurs (locaux et globaux)
- ▶ définition et utilisation de fonctions
- ▶ définition de types : synonyme, énuméré, produit, somme/union
- ▶ pattern-matching : `match ... with ...`
- ▶ **récurtivité**
 - ▶ fonction récursives (spécification, équations récursives, terminaison)
 - ▶ **types récursifs**

Rappels sur “récursivité” et “types récurifs”

fonction réursive

- ▶ définie en “fonction d'elle-même” par des **équations de récurrence** (cas de base, cas récurifs) :
 $0! = 1$ et $\forall n > 0. n! = n \times (n - 1)!$
- ▶ permet de décrire des **suites de calcul de longueur arbitraire**
ex : (fact 5), (fact 10), etc.
- ▶ problème de **terminaison**

Rappels sur “récursivité” et “types récurifs”

fonction récursive

- ▶ définie en “fonction d'elle-même” par des **équations de récurrence** (cas de base, cas récurifs) :
 $0! = 1$ et $\forall n > 0. n! = n \times (n - 1)!$
- ▶ permet de décrire des **suites de calcul de longueur arbitraire**
ex : (fact 5), (fact 10), etc.
- ▶ problème de **terminaison**

Type récurif

- ▶ défini en “fonction de lui-même” ... (cas de base, cas récurifs) :
- ▶ permet de décrire des **données de taille arbitraire**
entiers de Peano, polynômes, ligne brisée, etc.
- ▶ problème de **terminaison** : type “bien fondés”

Définition d'un type récursif (exemples)

- ▶ un **entier de Peano** est soit l'entier zero, soit le successeur d'un **entier de Peano**

⇒

Deux constructeurs :

- ▶ l'entier **zéro**
- ▶ le **successeur** d'un entier de Peano

Définition d'un type récursif (exemples)

- ▶ un **entier de Peano** est soit l'entier zero, soit le successeur d'un **entier de Peano**

⇒ Deux constructeurs :

- ▶ l'entier **zéro**
- ▶ le **successeur** d'un entier de Peano

- ▶ une **ligne brisée** est soit un segment unitaire orienté, soit une **ligne brisée** prolongée par un segment unitaire orienté :

⇒ Deux constructeurs :

- ▶ le **segment unitaire** orienté
- ▶ la **prolongation** d'une ligne brisée par un segment unitaire orienté

Définition d'un type récursif (exemples)

- ▶ un **entier de Peano** est soit l'entier zero, soit le successeur d'un **entier de Peano**

⇒ Deux constructeurs :

- ▶ l'entier **zéro**
- ▶ le **successeur** d'un entier de Peano

- ▶ une **ligne brisée** est soit un segment unitaire orienté, soit une **ligne brisée** prolongée par un segment unitaire orienté :

⇒ Deux constructeurs :

- ▶ le **segment unitaire** orienté
- ▶ la **prolongation** d'une ligne brisée par un segment unitaire orienté

- ▶ un **polynôme** est soit un monôme soit l'addition d'un **polynôme** et d'un monôme ...

⇒ Deux constructeurs :

- ▶ un **monôme**
- ▶ l'**addition** d'une nonôme et d'un polynôme

Fonction (récursive) à paramètres de types récurifs ?

Exemples

- ▶ longueur d'une ligne brisée
- ▶ somme des éléments d'un ensemble d'entiers
- ▶ valeur d'un polynôme en un point

Une approche systématique

→ filtrage sur les constructeurs de type

- ▶ les équations de récurrence **se déduisent** de la définition du type
- ▶ terminaison : le nombre de constructeurs décroît à chaque appel récursif

En Caml :

```
type t = C1 of ... | C2 of ... | Cn of ...
let rec f (x:t) : ... =
  match x with
  | C1 (...) -> ...
  | C2 (...) -> ...
  | ...
  | Cn (...) -> ...
```


Un type récursif “incontournable” : le type **liste** (ou séquences)

Problème fréquent en programmation :
gérer “**globalement**” un nombre **arbitraire** de données de **même type**

Exemples :

- ▶ un ensemble de valeurs numériques
- ▶ un texte (= une suite de caractères)
- ▶ une image (= un ensemble d’objets graphiques, ou une suite de pixels)
- ▶ etc.

↔ utilisation d’un type **liste** (ou **séquence**)

Un type récursif “incontournable” : le type **liste** (ou séquences)

Problème fréquent en programmation :
gérer “**globalement**” un nombre **arbitraire** de données de **même type**

Exemples :

- ▶ un ensemble de valeurs numériques
- ▶ un texte (= une suite de caractères)
- ▶ une image (= un ensemble d’objets graphiques, ou une suite de pixels)
- ▶ etc.

↔ utilisation d’un type **liste** (ou **séquence**)

La notion de liste/séquence existe dans la plupart des langages de programmation :

- ▶ elle est utile dans de nombreux contextes
- ▶ elle nécessite des opérateurs spécifiques (prédéfinis ou sous forme de bibliothèques)

⇒ il existe un “**vrai**” type **liste** en OCaml

Comment définir une liste ...

Qu'est-ce qu'une liste ?

- ▶ une **suite finie** de valeurs de même type
- ▶ contenant un nombre **arbitraire** d'éléments
- ▶ fournis selon un **ordre** donné ($[5, 8, 4] \neq [8, 5, 4]$)

Comment définir une liste ...

Qu'est-ce qu'une liste ?

- ▶ une **suite finie** de valeurs de même type
- ▶ contenant un nombre **arbitraire** d'éléments
- ▶ fournis selon un **ordre** donné ($[5, 8, 4] \neq [8, 5, 4]$)

Définition récursive du type liste $Seq(E)$

Etant donné un ensemble E , l'ensemble des listes sur E est le plus petit ensemble \mathcal{L}_E t.q. :

1. \mathcal{L}_E contient la **liste vide**
2. si $l \in \mathcal{L}_E$ et $e \in E$ alors la liste obtenue par **ajout** de e à l appartient à \mathcal{L}_E

Comment définir une liste ...

Qu'est-ce qu'une liste ?

- ▶ une **suite finie** de valeurs de même type
- ▶ contenant un nombre **arbitraire** d'éléments
- ▶ fournis selon un **ordre** donné ($[5, 8, 4] \neq [8, 5, 4]$)

Définition récursive du type liste $Seq(E)$

Etant donné un ensemble E , l'ensemble des listes sur E est le plus petit ensemble \mathcal{L}_E t.q. :

1. \mathcal{L}_E contient la **liste vide**
2. si $l \in \mathcal{L}_E$ et $e \in E$ alors la liste obtenue par **ajout** de e à l appartient à \mathcal{L}_E

Le type `Liste` est donc un type (union) **récuratif**, par exemple :

1. la constante `Nil` représente **la liste vide**
2. le constructeur `Cons(e,l)` représente **l'ajout en tête** de e à l

Remarque On peut aussi choisir un constructeur "ajout en fin" ...



Exemple de définition d'un type liste en Caml

Etant donné un type t (défini par ailleurs) :

```
type list_de_t = Nil | Cons of t * list_de_t
```

La liste dont les éléments sont v_1, v_2, \dots, v_n (de type t) est notée :

```
Cons (v1, Cons (v2, ..., Cons (vn, Nil) ...))
```

Exemple de définition d'un type liste en Caml

Etant donné un type t (défini par ailleurs) :

```
type list_de_t = Nil | Cons of t * list_de_t
```

La liste dont les éléments sont v_1, v_2, \dots, v_n (de type t) est notée :

```
Cons (v1, Cons (v2, ..., Cons (vn, Nil) ...))
```

Remarque

- ▶ un élément de type liste est considéré comme une **valeur**
- ▶ il peut être utilisé en tant que **valeur** dans toute construction du langage
- ▶ l'opérateur d'égalité prend en compte l'ordre des éléments :

```
Cons (1, Cons (2, Nil)) <> Cons (2, Cons (1, Nil))
```



DEMO: Liste d'entiers

Exemple de définition d'un type liste en Caml

Etant donné un type t (défini par ailleurs) :

```
type list_de_t = Nil | Cons of t * list_de_t
```

La liste dont les éléments sont v_1, v_2, \dots, v_n (de type t) est notée :

$$\text{Cons } (v_1, \text{Cons } (v_2, \dots, \text{Cons } (v_n, \text{Nil}) \dots))$$

Remarque

- ▶ un élément de type liste est considéré comme une **valeur**
- ▶ il peut être utilisé en tant que **valeur** dans toute construction du langage
- ▶ l'opérateur d'égalité prend en compte l'ordre des éléments :

$$\text{Cons } (1, \text{Cons } (2, \text{Nil})) \neq \text{Cons } (2, \text{Cons } (1, \text{Nil}))$$


DEMO: Liste d'entiers

Remarque On définit de même des listes de booléens, réels, fonctions, etc.



Typage

Une règle fondamentale : tous les éléments d'une liste sont *de même type*

Toutes les règles de typage déjà vues s'appliquent également aux listes (construction `if...then...else`, pattern-matching, fonctions, etc.)

Typage

Une règle fondamentale : tous les éléments d'une liste sont *de même type*

Toutes les règles de typage déjà vues s'appliquent également aux listes (construction `if...then...else`, pattern-matching, fonctions, etc.)

Nous verrons (plus tard) que :

- ▶ il existe construction de type en Ocaml pour définir une liste
- ▶ le type

```
type list_de_t = Nil | Cons of t * list_de_t
```

est en fait le type `t list` de OCaml, défini pour tout type `t`

Exercice : définir une liste dont certains éléments sont entiers, d'autres booléens ?

Autres constructions possibles du type liste ?

En faisant varier le nombre et la position d'insertion des nouveaux éléments ...

Définir les types (récur­sifs) correspondants aux listes suivantes :

1. les listes qui ne sont **jamais vides** (contenant **au moins** un élément)

Autres constructions possibles du type liste ?

En faisant varier le nombre et la position d'insertion des nouveaux éléments ...

Définir les types (récur­sifs) correspondants aux listes suivantes :

1. les listes qui ne sont **jamais vides** (contenant **au moins** un élément)
2. les listes de **longueur impaire** ? dont **la longueur est multiple de 5** ?

Autres constructions possibles du type liste ?

En faisant varier le nombre et la position d'insertion des nouveaux éléments ...

Définir les types (récur­sifs) correspondants aux listes suivantes :

1. les listes qui ne sont **jamais vides** (contenant **au moins** un élément)
2. les listes de **longueur impaire** ? dont **la longueur est multiple de 5** ?
3. les listes **palindromes** sur $\{a, b\}$

Autres constructions possibles du type liste ?

En faisant varier le nombre et la position d'insertion des nouveaux éléments ...

Définir les types (récur­sifs) correspondants aux listes suivantes :

1. les listes qui ne sont **jamais vides** (contenant **au moins** un élément)
2. les listes de **longueur impaire** ? dont **la longueur est multiple de 5** ?
3. les listes **palindromes** sur $\{a, b\}$
4. les listes sur $\{a, b\}$ contenant autant de a que de b

Autres constructions possibles du type liste ?

En faisant varier le nombre et la position d'insertion des nouveaux éléments ...

Définir les types (récursifs) correspondants aux listes suivantes :

1. les listes qui ne sont **jamais vides** (contenant **au moins** un élément)
2. les listes de **longueur impaire** ? dont **la longueur est multiple de 5** ?
3. les listes **palindromes** sur $\{a, b\}$
4. les listes sur $\{a, b\}$ contenant autant de a que de b

Remarque le choix des constructeurs n'est pas anodin ...

⇒ conséquences sur l'écriture de fonctions sur le type liste



Retour sur le pattern-matching

Bonne nouvelle, il est aussi utilisable sur les listes !

pattern-matching (opérateur `match ... with ...`):

→ sélectionne une valeur selon la “forme” d’une expression donnée

- ▶ le choix de la valeur est défini par **filtrage**
- ▶ les filtres représentent des **ensembles de valeurs** possibles

Retour sur le pattern-matching

Bonne nouvelle, il est aussi utilisable sur les listes !

pattern-matching (opérateur `match ... with ...`):

→ sélectionne une valeur selon la “forme” d’une expression donnée

- ▶ le choix de la valeur est défini par **filtrage**
- ▶ les filtres représentent des **ensembles de valeurs** possibles

Plusieurs formes de filtrage sur les listes :

ensembles de valeurs attendues	filtre
liste vide	<code>Nil</code>
liste non vide	<code>Cons (_, l), Cons (_, _), Cons (e, l), Cons (e, _)</code>
liste à un seul élément	<code>Cons (e, Nil), Cons (_, Nil)</code>
liste non vide dont le premier élément est 42 (dans le cas d’une liste d’entiers)	<code>Cons (42, l)</code>
...	...

Remarque Les filtres “équivalents” diffèrent dans la manière de nommer les (sous-)expressions filtrées ... □

Quelques fonctions (simples) sur les listes d'entiers

```
type intlist = Nil | Cons of int * intlist
```

DEMO: Fonctions simples et exemples d'implémentations

Quelques fonctions (simples) sur les listes d'entiers

```
type intlist = Nil | Cons of int * intlist
```

DEMO: Fonctions simples et exemples d'implémentations

Exemple (Crée une liste "singleton" (un seul élément) - `singList`)

- ▶ Profil: `singList: int → intlist`
- ▶ Description/Sémantique: `singList n` est la liste singleton constituée de l'élément `n`
- ▶ Exemples: `singList 5 = Cons (5,Nil)`

Quelques fonctions (simples) sur les listes d'entiers

```
type intlist = Nil | Cons of int * intlist
```

DEMO: Fonctions simples et exemples d'implémentations

Exemple (Crée une liste "singleton" (un seul élément) - `singList`)

- ▶ Profil: `singList: int → intlist`
- ▶ Description/Sémantique: `singList n` est la liste singleton constituée de l'élément `n`
- ▶ Exemples: `singList 5 = Cons (5,Nil)`

Exemple (Tête de liste - `tete`)

- ▶ Profil: `tete: intlist → int`
- ▶ Description/Sémantique: `tete l` renvoie le 1er élément de la liste `l`, et un message d'erreur si la liste est vide.
- ▶ Exs: `tete (Cons (1,Nil)) = 1`, `tete Nil = "erreur : liste vide"`

Quelques fonctions (simples) sur les listes d'entiers

```
type intlist = Nil | Cons of int * intlist
```

DEMO: Fonctions simples et exemples d'implémentations

Exemple (Crée une liste "singleton" (un seul élément) - `singList`)

- ▶ Profil: `singList: int → intlist`
- ▶ Description/Sémantique: `singList n` est la liste singleton constituée de l'élément `n`
- ▶ Exemples: `singList 5 = Cons (5,Nil)`

Exemple (Tête de liste - `tete`)

- ▶ Profil: `tete: intlist → int`
- ▶ Description/Sémantique: `tete l` renvoie le 1er élément de la liste `l`, et un message d'erreur si la liste est vide.
- ▶ Exs: `tete (Cons (1,Nil)) = 1`, `tete Nil = "erreur : liste vide"`

Autres exemples :

- ▶ `fin`
- ▶ `la_tete_vaut_0`
- ▶ `second`

Fonctions récursives sur les listes

La plupart des calculs effectués sur les listes s'expriment sous forme récursive ... car les listes sont un type récursif ...

Fonctions récursives sur les listes

La plupart des calculs effectués sur les listes s'expriment sous forme récursive ... car les listes sont un type récursif ...

Corps d'une fonction récursive sur les listes

→ une analyse par cas qui “reflète” la structure de l'argument “liste” :

- a) traitement de la liste vide (Nil)
- b) traitement de la liste non-vide (Cons (elt,fin)):
le résultat dépend de l'élément courant `elt` et du résultat d'appel(s) récursif(s) sur `fin`.

Fonctions récursives sur les listes

La plupart des calculs effectués sur les listes s'expriment sous forme récursive ... car les listes sont un type récursif ...

Corps d'une fonction récursive sur les listes

→ une analyse par cas qui “reflète” la structure de l'argument “liste” :

- a) traitement de la liste vide (Nil)
- b) traitement de la liste non-vide (Cons (elt,fin)):
le résultat dépend de l'élément courant `elt` et du résultat d'appel(s) récursif(s) sur `fin`.

Pour définir une fonction récursive $f: \text{list_de_}t1 \rightarrow t2$,

- a) $f \text{ Nil} = \dots$ une valeur dans $t2$...
- b) $f (\text{Cons } (elt, fin)) = (g (h \text{ elt}) (f \text{ fin}))$
avec $h: t1 \rightarrow t3$ and $g: t3 \rightarrow t2 \rightarrow t2$

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: `longueur: intlist → int`
- ▶ Sémantique: `longueur l = |l|`, le nombre d'éléments
- ▶ Exemples: `longueur Nil=0`, `longueur (Cons(9,Nil))=1...`
- ▶ Equations récursives :

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: `longueur: intlist → int`
- ▶ Sémantique: `longueur l = |l|`, le nombre d'éléments
- ▶ Exemples: `longueur Nil=0`, `longueur (Cons(9,Nil))=1...`
- ▶ Equations récursives :

`longueur Nil =`

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: `longueur: intlist → int`
- ▶ Sémantique: `longueur l = |l|`, le nombre d'éléments
- ▶ Exemples: `longueur Nil=0`, `longueur (Cons(9,Nil))=1...`
- ▶ Equations récursives :

$$\begin{aligned} \text{longueur } Nil &= 0 \\ \text{longueur } (Cons(a, l)) &= \end{aligned}$$

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: `longueur: intlist → int`
- ▶ Sémantique: `longueur l = |l|`, le nombre d'éléments
- ▶ Exemples: `longueur Nil=0`, `longueur (Cons(9,Nil))=1...`
- ▶ Equations récursives :

$$\begin{aligned} \text{longueur } Nil &= 0 \\ \text{longueur } (Cons(a, l)) &= 1 + \text{longueur } l \end{aligned}$$

- ▶ Terminaison ?

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: $\text{longueur} : \text{intlist} \rightarrow \text{int}$
- ▶ Sémantique: $\text{longueur } l = |l|$, le nombre d'éléments
- ▶ Exemples: $\text{longueur Nil} = 0$, $\text{longueur (Cons(9,Nil))} = 1 \dots$
- ▶ Equations récursives :

$$\begin{aligned}\text{longueur } Nil &= 0 \\ \text{longueur } (Cons(a, l)) &= 1 + \text{longueur } l\end{aligned}$$

- ▶ Terminaison ?
 - ▶ Soit $\text{mesure}(\text{longueur } l) = \text{taille}(l)$ où $\text{taille}(l)$ est le nbre d'occurrences du constructeur `Cons` dans l
 - ▶ Nous avons : $\text{mesure}(\text{longueur } Cons(_, l)) > \text{mesure}(\text{longueur } l)$

Définition de fonctions récursives sur les listes

Longueur d'une liste

La longueur d'une liste est le nombre de ses éléments

- ▶ Profil: longueur: intlist \rightarrow int
- ▶ Sémantique: longueur l = |l|, le nombre d'éléments
- ▶ Exemples: longueur Nil=0, longueur (Cons(9,Nil))=1...
- ▶ Equations récursives :

$$\begin{aligned} \text{longueur } Nil &= 0 \\ \text{longueur } (Cons(a, l)) &= 1 + \text{longueur } l \end{aligned}$$

- ▶ Terminaison ?
 - ▶ Soit mesure(longueur l) = taille(l) où taille(l) est le nbre d'occurrences du constructeur Cons dans l
 - ▶ Nous avons : mesure(longueur Cons(_,l)) > mesure(longueur l)
- ▶ Implémentation

```
let rec longueur (l:intlist):int=  
  match l with  
  | Nil  $\rightarrow$  0  
  | Cons (_,l)  $\rightarrow$  1+longueur l
```

DEMO: Exemple d'exécution sur Cons(1,Cons(2,Nil))

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction (“accepter” le *warning* émis par l’interpréteur)

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction (“accepter” le *warning* émis par l’interpréteur)
2. générer une exception en cas d’appel avec une liste vide (en utilisant `failwith` ou `assert`)

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction (“accepter” le *warning* émis par l’interpréteur)
2. générer une exception en cas d’appel avec une liste vide (en utilisant `failwith` ou `assert`)
3. définir et utiliser un type spécifique : le type *liste non-vide*

```
type intlist_non_vide=  
  Elt of int  
  | Cons of int * intlist_non_vide
```

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction (“accepter” le *warning* émis par l’interpréteur)
2. générer une exception en cas d’appel avec une liste vide (en utilisant `failwith` ou `assert`)
3. définir et utiliser un type spécifique : le type *liste non-vide*

```
type intlist_non_vide=  
  Elt of int  
  | Cons of int * intlist_non_vide
```

4. renvoyer un booléen en plus du résultat indiquant si celui-ci est *pertinent*
↔ l’utilisation du résultat est conditionnée par ce booléen ...

Cas des fonctions **non définies sur les listes vides**

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction (“accepter” le *warning* émis par l’interpréteur)
2. générer une exception en cas d’appel avec une liste vide (en utilisant `failwith` ou `assert`)
3. définir et utiliser un type spécifique : le type *liste non-vide*

```
type intlist_non_vide=  
  Elt of int  
  | Cons of int * intlist_non_vide
```

4. renvoyer un booléen en plus du résultat indiquant si celui-ci est *pertinent*
↔ l’utilisation du résultat est conditionnée par ce booléen ...
5. définir une fonction à deux paramètres :
le premier element de la liste et la fin de la liste (éventuellement vide).

Exercices

Pour des listes d'entiers :

- ▶ `somme` : renvoie la somme des éléments d'une liste
- ▶ `appartient` : indique si un élément appartient à une liste
- ▶ `dernier` : renvoie le dernier élément d'une liste
- ▶ `minimum` : renvoie le minimum d'une liste d'entiers
- ▶ `pairs` : renvoie les entiers pairs d'une liste
- ▶ `supprime` : supprime une/toutes les occurrences d'un élément
- ▶ `concatene` : concaténation de deux listes
- ▶ `est_croissante` : indique si une liste est croissante

Exercices

Pour des listes d'entiers :

- ▶ `somme` : renvoie la somme des éléments d'une liste
- ▶ `appartient` : indique si un élément appartient à une liste
- ▶ `dernier` : renvoie le dernier élément d'une liste
- ▶ `minimum` : renvoie le minimum d'une liste d'entiers
- ▶ `pairs` : renvoie les entiers pairs d'une liste
- ▶ `supprime` : supprime une/toutes les occurrences d'un élément
- ▶ `concatene` : concaténation de deux listes
- ▶ `est_croissante` : indique si une liste est croissante

Remarque mêmes fonctions avec des listes construite par "ajout en queue"



Une liste de cartes

- ▶ Définir (à l'aide d'un type liste) une “*main*” contenant 5 cartes
- ▶ Ecrire une fonction qui calcule la valeur de cette *main*

Exemple (Liste de cartes)

```
type carte = Petite of int | Valet | Dame | Cavalier | Roi | As
type main = Nil | Cons of carte * main
```

- ▶ valeur_carte: carte → int
- ▶ valeur_main: main → int

Conclusion (provisoire) sur les listes

- ▶ un type récursif “incontournable”
- ▶ permet de définir d'autres types (ensemble, polynômes, etc.)
- ▶ différents modes de construction (ajout en tête, en queue, etc.)
→ à adapter aux traitements envisagés ...
- ▶ il existe un type prédéfini `list` en Ocaml
 - ▶ construction par ajout en tête
 - ▶ fonctions prédéfinies (bibliothèque)
- ▶ on peut s'aider de **schémas génériques** pour écrire des fonctions sur des listes ...