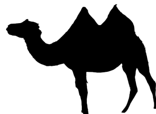
GRENOBLE 1
UNIVERSITE JOSEPH FOURIER
SCIENCES.TECHNOLOGIE.SANTÉ

INF201
Algorithmique et Programmation Fonctionnelle
Cours 7 : Listes (suite)

Année 2017 - 2018



Brefs rappels sur les listes

Une liste c'est :

- une suite finie, ordonnée, de valeurs de même type
- pouvant contenir un nombre arbitraire d'éléments

Brefs rappels sur les listes

Une liste c'est :

- une suite finie, ordonnée, de valeurs de même type
- pouvant contenir un nombre arbitraire d'éléments

Le type `liste_de_t` est défini par un type (union) récursif :

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

`Cons` (`expr1`, `Cons` (`expr2`, ...`Cons` (`exprn`, `Nil`) ...))

Brefs rappels sur les listes

Une liste c'est :

- une suite finie, ordonnée, de valeurs de même type
- pouvant contenir un nombre arbitraire d'éléments

Le type `liste_de_t` est défini par un type (union) récursif :

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

`Cons` (`expr1`, `Cons` (`expr2`, ...`Cons` (`exprn`, `Nil`) ...))

→ Un élément du type `list_de_t` est une valeur (comme une autre !)

→ On peut définir des listes d'entiers, booléens, réels, fonctions, etc.

→ On peut considérer d'autres constructeurs, selon la "forme" des listes souhaitées (ajout en queue, listes non vides, de longueur impaire, etc.)

Le type prédéfini pour les listes en Ocaml

OCaml fournit un constructeur de type `list` prédéfini

- `Nil` est noté [ ]
- `Cons` est exprimé par un opérateur infixé ::

Le type prédéfini pour les listes en Ocaml

OCaml fournit un constructeur de type `list` prédéfini

- `Nil` est noté [ ]
- `Cons` est exprimé par un opérateur infixé ::

Exemple : Listes en notation OCaml

- `Cons` (2, `Nil`) correspond à 2::[ ]
- `Cons` (4,`Cons` (9, `Nil`)) correspond à 4::(9::[ ])

Le type prédéfini pour les listes en Ocaml

OCaml fournit un constructeur de type `list` prédéfini

- `Nil` est noté [ ]
- `Cons` est exprimé par un opérateur infixé ::

Exemple : Listes en notation OCaml

- `Cons` (2, `Nil`) correspond à 2::[ ]
- `Cons` (4,`Cons` (9, `Nil`)) correspond à 4::(9::[ ])

Quelques raccourcis de notation :

- `v1`::(`v2`::...::(`vn`::[ ])) peut être noté `v1`::`v2`:: ...`vn`::[ ]
- `v1`::`v2`::... `vn`::[ ] peut être noté [`v1`;`v2`;...;`vn`]

Exemple : `Cons` (4,`Cons` (9, `Cons` (5, `Nil`))) se note [4;9;5]

Le type prédéfini pour les listes en Ocaml

OCaml fournit un constructeur de type `list` prédéfini

- `Nil` est noté [ ]
- `Cons` est exprimé par un opérateur infixé ::

Exemple : Listes en notation OCaml

- `Cons` (2, `Nil`) correspond à 2::[ ]
- `Cons` (4,`Cons` (9, `Nil`)) correspond à 4::(9::[ ])

Quelques raccourcis de notation :

- `v1`::(`v2`::...::(`vn`::[ ])) peut être noté `v1`::`v2`:: ...`vn`::[ ]
- `v1`::`v2`::... `vn`::[ ] peut être noté [`v1`;`v2`;...;`vn`]

Exemple : `Cons` (4,`Cons` (9, `Cons` (5, `Nil`))) se note [4;9;5]

Le type “liste d'éléments de type `t`” se note `t list`

DEMO: liste d'entiers, de couples, de couleurs, de listes, etc.

Retour sur le pattern-matching

Rien ne bouge . . .

Comment effectuer un traitement par cas en “filtrant” certaines formes de listes ?

Retour sur le pattern-matching

Rien ne bouge ...

Comment effectuer un traitement par cas en “filtrant” certaines formes de listes ?

⇒ comme pour un type récursif à 2 constructeurs `Cons` et `Nil`,
mais avec une nouvelle syntaxe :

ensembles de valeurs attendues	**filtre**
liste vide	[]
liste non vide	`_::_`, `_::fin`, `e::fin`, `e::_`
liste à un seul élément	`e::[]`, `_::[]`, `[_]`, `[e]`
liste non vide dont le premier élément est 0	`0::_`
liste à au moins deux élément	`e1::e2::fin`, `e::_::_`
...	...

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow t$?
(exemple : le **dernier** élément d'une liste d'entier)

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow t$?
(exemple : le **dernier** élément d'une liste d'entier)

Plusieurs alternatives :

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow t$?
(exemple : le **dernier** élément d'une liste d'entier)

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction
   ("accepter" le *warning* émis par l'interpréteur)
   $\rightarrow$ on suppose que l'appelant respectera la spécification . . .

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow$ t ?
(exemple : le **dernier** élément d'une liste d'entier)

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction
   ("accepter" le *warning* émis par l'interpréteur)
   $\rightarrow$ on suppose que l'appelant respectera la spécification . . .

2. générer une exception en cas d'appel avec une liste vide
   (en utilisant `failwith` ou `assert`)

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow t$?
(exemple : le **dernier** élément d'une liste d'entier)

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction
   (“accepter” le *warning* émis par l'interpréteur)
   $\rightarrow$ on suppose que l'appelant respectera la spécification . . .
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
```

Et les listes **non vides** ?

Pas de type CAML prédédini . . .

Ecrire une fonction f telle que f : liste non vide d'entiers $\rightarrow t$?
(exemple : le **dernier** élément d'une liste d'entier)

Plusieurs alternatives :

1. ne pas considérer le cas où la liste est vide dans la fonction
   ("accepter" le *warning* émis par l'interpréteur)
   $\rightarrow$ on suppose que l'appelant respectera la spécification . . .
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*
   $\hookrightarrow$ l'utilisation du résultat est conditionnée par ce booléen . . .

Retour sur les exercices du cours précédent

Avec ces nouvelles notations

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 la liste des entiers pairs d'une liste d'entiers
  (variante : renvoie la liste des entiers de rang pair)
- `supprime` : supprime une/toutes les occurrences d'un élément
- `concatene` : concaténation de deux listes
- `partition` : partitionne une liste en deux sous-listes selon un critère donné
- `est_croissante` : indique si une liste est croissante

DEMO: Implémentation de certaines de ces fonctions

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

$\rightarrow$ Pour écrire une fonction portant sur un type récursif, on peut s’aider de la définition de ce type !

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

→ Pour écrire une fonction portant sur un type récursif, on peut s'aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f` : `t1 list` → `t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, . . .

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

→ Pour écrire une fonction portant sur un type récursif, on peut s’aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f` : `t1 list` → `t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, . . .

1. **Cas de base :** valeur de `f` pour la liste vide : `f`([]) = ...

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents ...

$\rightarrow$ Pour écrire une fonction portant sur un type récursif, on peut s’aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f` : `t1 list` $\rightarrow$ `t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, ...

1. **Cas de base :** valeur de `f` pour la liste vide : `f`([]) = ...
2. Exprimer `f` sous **forme récursive** $\rightarrow$ découper la liste en isolant :
   - son premier élément `e` et son suffixe `s` :

     `f(e::s) = (g e (f s))`

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

→ Pour écrire une fonction portant sur un type récursif, on peut s’aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f : t1 list → t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, . . .

1. **Cas de base :** valeur de `f` pour la liste vide : `f([]) = ...`
2. Exprimer `f` sous **forme récursive** → découper la liste en isolant :
   - son premier élément `e` et son suffixe `s` :

     `f(e::s) = (g e (f s))`

   - ou ses deux premiers éléments `e1`, `e2` et son suffixe `s` :

```
f([e1])        =   ...  (cas de la liste à un element)
f(e1::e2::s)   =   (g e1 e2 (f (e2::s)))
```

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

→ Pour écrire une fonction portant sur un type récursif, on peut s’aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f : t1 list → t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, . . .

1. **Cas de base :** valeur de `f` pour la liste vide : `f([]) = ...`
2. Exprimer `f` sous **forme récursive** → découper la liste en isolant :
   - son premier élément `e` et son suffixe `s` :

     `f(e::s) = (g e (f s))`

   - ou ses deux premiers éléments `e1`, `e2` et son suffixe `s` :

     ```
     f([e1])       =   ...  (cas de la liste à un element)
     f(e1::e2::s)  =   (g e1 e2 (f (e2::s)))
     ```

   - etc.

Comment écrire une fonction à paramètres de type liste ?

Quelques “schémas” assez fréquents . . .

→ Pour écrire une fonction portant sur un type récursif, on peut s'aider de la définition de ce type !

Fonction à 1 seul paramètre `l` de type liste (`f : t1 list → t2`)

exemples : longueur, appartient, partition, pair, supprime, est_croissant, . . .

1. **Cas de base :** valeur de `f` pour la liste vide : `f([]) = ...`
2. Exprimer `f` sous **forme récursive** → découper la liste en isolant :
   - son premier élément `e` et son suffixe `s` :

     `f(e::s) = (g e (f s))`

   - ou ses deux premiers éléments `e1`, `e2` et son suffixe `s` :

     ```
     f([e1])       =   ...  (cas de la liste à un element)
     f(e1::e2::s)  =   (g e1 e2 (f (e2::s)))
     ```

   - etc.

Implémentation en CAML par filtrage :

```
let rec somme (l : int list) : int =
 match l with
  [] → 0  (* liste vide *)
  | e::s → e + (somme s) (* liste non vide *)
```

Comment écrire une fonction à paramètres de type liste (suite) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)
exemple : concaténation

Approche 1 : on raisonne sur les 2 liste → 4 cas possibles

Comment écrire une fonction à paramètres de type liste (suite) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)

exemple : concaténation

Approche 1 : on raisonne sur les 2 liste → 4 cas possibles

1. valeur de `f` pour les deux liste vide : `f`([], []) = ...

Comment écrire une fonction à paramètres de type liste (suite) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)

exemple : concaténation

Approche 1 : on raisonne sur les 2 liste → 4 cas possibles

1. valeur de `f` pour les deux liste vide : `f`([], []) = ...
2. valeur de f pour une liste vide, l'autre non vide

   `f`(`l1`, []) = ... et `f`([], `l2`) = ...

Comment écrire une fonction à paramètres de type liste (suite) ?

Fonction à 2 paramètres de type liste (`f : t1 list` $\rightarrow$ `t2 list` $\rightarrow$ `t3`)

exemple : concaténation

Approche 1 : on raisonne sur les 2 liste $\rightarrow$ 4 cas possibles

1. valeur de `f` pour les deux liste vide : `f`([], []) = ...
2. valeur de f pour une liste vide, l'autre non vide

   `f`(`l1`, []) = ... et `f`([], `l2`) = ...

3. valeur de f pour les deux listes non vides (cas récursif)
   $\rightarrow$ découpage de chaque liste en premier(s) élément(s) et suffixes

   `f`(`e1`::`s1`, `e2`::`s2`) = ...

Comment écrire une fonction à paramètres de type liste (suite) ?

Fonction à 2 paramètres de type liste (`f : t1 list → t2 list → t3`)

exemple : concaténation

Approche 1 : on raisonne sur les 2 liste → 4 cas possibles

1. valeur de `f` pour les deux liste vide : `f([], [])` = ...
2. valeur de f pour une liste vide, l'autre non vide

   `f(l1, [])` = ... et `f([], l2)` = ...

3. valeur de f pour les deux listes non vides (cas récursif)
   → découpage de chaque liste en premier(s) élément(s) et suffixes

   `f(e1::s1, e2::s2)` = ...

```
let rec concat (l1 : t list) (l2 : t list) : t list =
 match l1, l2 with
   [], [] → []
 | [], l2 → l2
 | l1, [] → l1
 | e1::s1, e2::s2 → e1::(concat s1 (e2::s2))
```

Comment écrire une fonction à paramètres de type liste (fin) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)

exemple : concaténation

Approche 2 : on raisonne sur une seule des deux listes (p. ex. `l1`)

Comment écrire une fonction à paramètres de type liste (fin) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)

exemple : concaténation

Approche 2 : on raisonne sur une seule des deux listes (p. ex. `l1`)

1. **cas de base :** valeur de `f` lorsque `l1` est vide

$$f([], l2) = \ldots$$

Comment écrire une fonction à paramètres de type liste (fin) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)

exemple : concaténation

Approche 2 : on raisonne sur une seule des deux listes (p. ex. `l1`)

1. **cas de base :** valeur de `f` lorsque `l1` est vide

   `f([], l2) = ...`

2. **cas récursifs :** on découpe `l1` en premier(s) élément(s) et suffixe

   `f(e1::s1, l2) = ...`

   Rq : on peut être amené à distinguer plusieurs cas en fonction de `l2`.

Comment écrire une fonction à paramètres de type liste (fin) ?

Fonction à 2 paramètres de type liste (`f` : `t1 list` → `t2 list` → `t3`)
exemple : concaténation

Approche 2 : on raisonne sur une seule des deux listes (p. ex. `l1`)

1. **cas de base :** valeur de `f` lorsque `l1` est vide

   `f([], l2) = ...`

2. **cas récursifs :** on découpe `l1` en premier(s) élément(s) et suffixe

   `f(e1::s1, l2) = ...`

   Rq : on peut être amené à distinguer plusieurs cas en fonction de `l2`.

Implémentation en CAML par filtrage :

```
let rec concat (l1 : t list) (l2 : t list) : t list =
 match l1 with
   [] →  l2
   | e1::s1 → e1::(concat s1 l2)
```

Liste génériques

Certaines fonctions sur les listes ne dépendent pas du type des éléments ...

Exemples :

- longueur, dernier, appartient, supprime, concatene, ...
- minimum, est_croissante, partition, ...
  (en considérant l'ordre induit par l'opérateur <)

Liste génériques

Certaines fonctions sur les listes ne dépendent pas du type des éléments ...

Exemples :

- longueur, dernier, appartient, supprime, concatene, ...
- minimum, est_croissante, partition, ...
  (en considérant l'ordre induit par l'opérateur <)

$\Rightarrow$ on peut définir des listes génériques (à élts de type "quelconque")

notation CAML : `'a list` (se prononce "α-liste")

Liste génériques

Certaines fonctions sur les listes ne dépendent pas du type des éléments ...

Exemples :

- longueur, dernier, appartient, supprime, concatene, ...
- minimum, est_croissante, partition, ...
  (en considérant l'ordre induit par l'opérateur <)

$\Rightarrow$ on peut définir des listes génériques (à élts de type "quelconque")

notation CAML : `'a list` (se prononce "α-liste")

Exemple :
Appartenance d'un élément à une liste générique

```
let rec appartient (x: 'a)(l : 'a list) : bool =
  match l with
      [] -> false
     |e::s -> e=x || (appartient x s)
```

Liste génériques

Certaines fonctions sur les listes ne dépendent pas du type des éléments ...

Exemples :

- longueur, dernier, appartient, supprime, concatene, ...
- minimum, est_croissante, partition, ...
  (en considérant l'ordre induit par l'opérateur <)

$\Rightarrow$ on peut définir des listes génériques (à élts de type "quelconque")

notation CAML : `'a list` (se prononce "α-liste")

Exemple :
Appartenance d'un élément à une liste générique

```
let rec appartient (x: 'a)(l : 'a list) : bool =
  match l with
     [] -> false
     |e::s -> e=x || (appartient x s)
```

(`appartient` 2 [5;8;4;5;2;1]) = `true`
(`appartient` `true` [`false`;`false`;`true`]) = `true`
(`appartient` `'u'` [`'z'`;`'x'`;`'w'`]) = `false`

Exercices

Fermeture-Eclair

`zip`: prend en paramètre un couple de liste (de même taille) et renvoie la liste des couples correspondants

Exemples :

- `zip` [1; 3; 8] [2; 9; 8] est la liste [(1,2); (3,9); (8,8)]
- `zip` [1; 3; 8; 10] [2; 9] n'est pas défini ...
  variante : compléter avec le dernier élt de la liste la plus courte
  `zip` [1; 3; 8 ; 10] [2; 9] = [(1,2) ; (3, 9) ; (8, 9) ; (10, 9)]

Exercices

Fermeture-Eclair

`zip`: prend en paramètre un couple de liste (de même taille) et renvoie la liste des couples correspondants

Exemples :

- `zip` [1; 3; 8] [2; 9; 8] est la liste [(1,2); (3,9); (8,8)]
- `zip` [1; 3; 8; 10] [2; 9] n'est pas défini . . .
  variante : compléter avec le dernier élt de la liste la plus courte
  `zip` [1; 3; 8 ; 10] [2; 9] = [(1,2) ; (3, 9) ; (8, 9) ; (10, 9)]

Une liste est-elle une sous-liste d'une autre ?

Exemples :

- [`e2` ; `e4` ; `e5`] est une sous-liste de [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e2` ; `e4` ; `e5` ; `e7`] n'est pas une sous-liste de [`e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e4` ; `e2` ; `e5`] n'est pas une sous-liste de [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]

Exercices

Fermeture-Eclair

`zip`: prend en paramètre un couple de liste (de même taille) et renvoie la liste des couples correspondants

Exemples :

- `zip` [1; 3; 8] [2; 9; 8] est la liste [(1,2); (3,9); (8,8)]
- `zip` [1; 3; 8; 10] [2; 9] n'est pas défini ...
  variante : compléter avec le dernier élt de la liste la plus courte
  `zip` [1; 3; 8 ; 10] [2; 9] = [(1,2) ; (3, 9) ; (8, 9) ; (10, 9)]

Une liste est-elle une sous-liste d'une autre ?

Exemples :

- [`e2` ; `e4` ; `e5`] est une sous-liste de [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e2` ; `e4` ; `e5` ; `e7`] n'est pas une sous-liste de [`e2` ; `e3` ; `e4` ; `e5` ; `e6`]
- [`e4` ; `e2` ; `e5`] n'est pas une sous-liste de [`e1` ; `e2` ; `e3` ; `e4` ; `e5` ; `e6`]

Analyse :

- ce predicat prend deux listes *l1* et *l2* en paramètre
- *l2* doit être obtenue en "supprimant" certains éléments de *l1*

Quelques opérateurs prédéfinis sur le type OCaml `list`

`let l1` = [3; 8; 7; 1]

Description	Opérateur	Exemple
nième élément	`List.nth`	`List.nth l1 2 = 7`
longueur	`List.length`	`List.length l1 = 4`
tête	`List.hd`	`List.hd l1 = 4`
fin	`List.tl`	`List.tl l1 = [8;7;1]`
inverse	`List.rev`	`List.rev l1 = [1;7;8;3]`
concaténation	`@`	`l1 @ [2;3] = [3; 8; 7; 1; 2; 3]`

Plus de détails sur

`caml.inria.fr/pub/docs/manual-ocaml/libref/List.html`

Listes, Ensembles et Multi-ensembles . . .

Trois notions à ne pas confondre !

Listes, Ensembles et Multi-ensembles ...

Trois notions à ne pas confondre !

Liste

- les éléments peuvent être dupliqués : [1;9;7] $\neq$ [1;9;9;7]
- l'ordre des éléments est important : [1;9;7] $\neq$ [1;7;9]
- implémentation directe à l'aide du type `list` ...

Listes, Ensembles et Multi-ensembles ...

Trois notions à ne pas confondre !

Liste

- les éléments peuvent être dupliqués : [1;9;7] $\neq$ [1;9;9;7]
- l'ordre des éléments est important : [1;9;7] $\neq$ [1;7;9]
- implémentation directe à l'aide du type `list` ...

Ensemble

- les éléments **ne sont pas** dupliqués : $\{1, 9, 7\} = \{1, 9, 9, 7\}$
- l'ordre des éléments ne compte pas : $\{1, 9, 7\} = \{1, 7, 9\}$
- implémentation possible à l'aide du type `list`, **mais** :
  - ne pas dupliquer les éléments (ou définir correctement la supression !)
  - ordonner les éléments (ou définir correctement l'égalité !)

Listes, Ensembles et Multi-ensembles ...

Trois notions à ne pas confondre !

Liste

- les éléments peuvent être dupliqués : [1;9;7] $\neq$ [1;9;9;7]
- l'ordre des éléments est important : [1;9;7] $\neq$ [1;7;9]
- implémentation directe à l'aide du type `list` ...

Ensemble

- les éléments **ne sont pas** dupliqués : $\{1,9,7\} = \{1,9,9,7\}$
- l'ordre des éléments ne compte pas : $\{1,9,7\} = \{1,7,9\}$
- implémentation possible à l'aide du type `list`, **mais** :
  - ne pas dupliquer les éléments (ou définir correctement la supression !)
  - ordonner les éléments (ou définir correctement l'égalité !)

Multi-ensemble

- les éléments peuvent être dupliqués : $\{1,9,7\} \neq \{1,9,9,7\}$
- l'ordre des éléments ne compte pas : $\{1,9,7\} = \{1,7,9\}$
- implémentation possible à l'aide du type `list`, **mais** :
  - dupliquer les éléments, ou utiliser des couples (elt, nbre_occurrence)
  - ordonner les éléments (ou définir correctement l'égalité !)

Tris de listes

Motivations

Trier $\approx$ ranger les éléments d'une liste selon un ordre donné :

$$\texttt{tri}\ (\texttt{l} : \alpha\ \texttt{list}) : \alpha\ \texttt{list}$$

$$\text{liste quelconque} \xrightarrow{\text{tri}} \text{liste triée selon la relation} <$$

Trier ≈ ranger les éléments d'une liste selon un ordre donné :

$$\texttt{tri}\ (\texttt{l} : \alpha\ \texttt{list}) : \alpha\ \texttt{list}$$

$$\text{liste quelconque} \xrightarrow{\text{tri}} \text{liste triée selon la relation} <$$

Exemple

$[2;1;9;4] \xrightarrow{\text{tri}} [1;2;4;9]$

- `type person = Toto | Titi | Tata`
- `[Titi;Tata;Toto]` $\xrightarrow{\text{tri}}$ `[Toto;Titi;Tata]`

Trier ≈ ranger les éléments d'une liste selon un ordre donné :

$$\texttt{tri}\ (\texttt{l} : \alpha\ \texttt{list}) : \alpha\ \texttt{list}$$

$$\text{liste quelconque} \xrightarrow{\text{tri}} \text{liste triée selon la relation} <$$

Exemple

$[2;1;9;4] \xrightarrow{\text{tri}} [1;2;4;9]$

- `type person = Toto | Titi | Tata`
- `[Titi;Tata;Toto]` $\xrightarrow{\text{tri}}$ `[Toto;Titi;Tata]`

Motivations ?

- **représentation canonique** du (multi-)ensemble des elts d'une liste
- certains traitements seront plus **efficaces**
- une étape nécessaire dans de nombreux algorithmes **plus généraux**

Trier $\approx$ ranger les éléments d'une liste selon un ordre donné :

$$\texttt{tri}\,(\texttt{l} : \alpha\ \texttt{list}) : \alpha\ \texttt{list}$$

$$\text{liste quelconque} \xrightarrow{\text{tri}} \text{liste triée selon la relation} <$$

Exemple

[2;1;9;4] $\xrightarrow{\text{tri}}$ [1;2;4;9]

- `type person = Toto | Titi | Tata`
- `[Titi;Tata;Toto]` $\xrightarrow{\text{tri}}$ `[Toto;Titi;Tata]`

Motivations ?

- **représentation canonique** du (multi-)ensemble des elts d'une liste
- certains traitements seront plus **efficaces**
- une étape nécessaire dans de nombreux algorithmes **plus généraux**

$\Rightarrow \exists$ nombreux **algorithmes de tri**, qui diffèrent par :

- leur efficacité (en temps d'exécution, en mémoire)
- leur difficulté à programmer, à prouver

Exemple 1 : tri par insertion (`tri_insertion (l : `α` list) : `α` list`)

Algorithme

- si `l`=[], le résultat est la liste vide ...

Exemple 1 : tri par insertion (`tri_insertion (l : α list) : α list`)

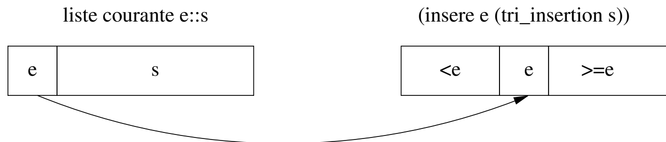
Algorithme

- si `l=[]`, le résultat est la liste vide ...
- si `l=e::s` ?

Exemple 1 : tri par insertion (`tri_insertion (l : α list) : α list`)

Algorithme

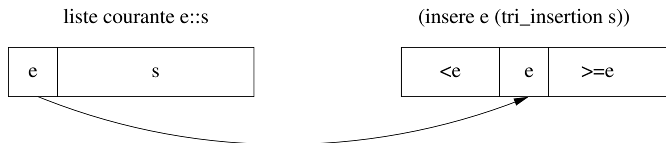
- si `l`=[], le résultat est la liste vide ...
- si `l=e::s` ?
  trier (**récursivement !**) `s` et insérer `e` (**à sa place !**) dans la liste résultat



Exemple 1 : tri par insertion (`tri_insertion (l : α list) : α list`)

Algorithme

- si `l`=[], le résultat est la liste vide ...
- si `l=e::s` ?
  trier (**récursivement !**) `s` et insérer `e` (**à sa place !**) dans la liste résultat

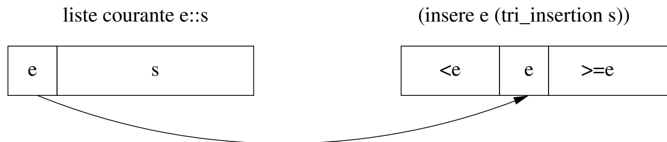


Insérer un élément à sa place dans une liste triée ?

Exemple 1 : tri par insertion (`tri_insertion (l : α list) : α list`)

Algorithme

- si `l=[]`, le résultat est la liste vide ...
- si `l=e::s` ?
  trier (**récursivement !**) `s` et insérer `e` (**à sa place !**) dans la liste résultat



Insérer un élément à sa place dans une liste triée ?

```
let rec inserer (e:α) (l:α list):α list=
(* l est croissante, le resultat est croissante *)
 match l with
   | [] → [e]
   | x::s → if e<x then e::l else x::(inserer e s)
```

DEMO: tri par insertion

Exemple 2 : tri par sélection (`tri_selection` (`l` : α `list`) : α `list`)

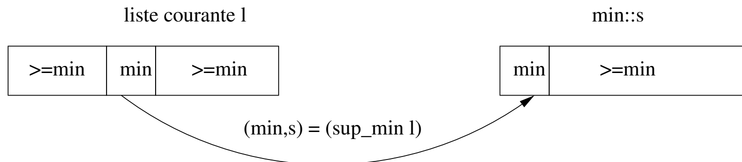
Algorithme

- si `l`=[], le résultat est la liste vide . . .
- sinon ?

Exemple 2 : tri par sélection (`tri_selection (l : α list) : α list`)

Algorithme

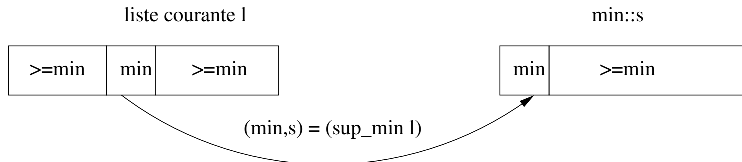
- si `l`=[], le résultat est la liste vide . . .
- sinon ?
  - soit `min` le minimum de `l` et `s` la liste `l` privée de `min`
  - on ajoute `min` (**en tête** !) de la liste `s` triée (**récursivement !**)



Exemple 2 : tri par sélection (`tri_selection (l : α list) : α list`)

Algorithme

- si `l`=[], le résultat est la liste vide . . .
- sinon ?
  - soit `min` le minimum de `l` et `s` la liste `l` privée de `min`
  - on ajoute `min` (**en tête** !) de la liste `s` triée (**récursivement !**)



Extraire d'une liste son minimum et le reste de la liste ?

`sup_min` : α `list` $\rightarrow$ ($\alpha * \alpha$ `list`)

`sup_min` (`l`) renvoie (`min`,`s`) tel que `min` est un élément minimum de la liste non vide `l` et `s` est la liste `l` privée de `min`.

DEMO: tri par sélection

Exemple 3 : tri rapide (`tri_rapide` (`l` : α `list`) : α `list`)

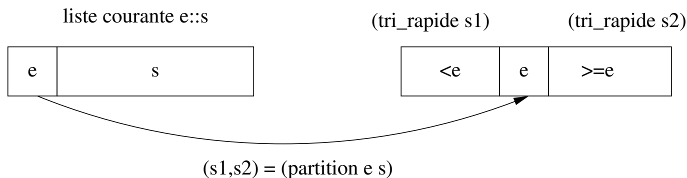
Algorithme

- si `l`=[], le résultat est la liste vide ...
- si `l`=`e`::`s` alors

Exemple 3 : tri rapide (`tri_rapide (l : α list) : α list`)

Algorithme

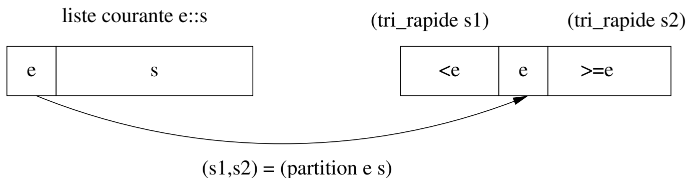
- si `l=[]`, le résultat est la liste vide ...
- si `l=e::s` alors
  - on partitionne `s` en `s1` (elts de `s` $<$ `e`) et `s2` (elts de `s` $\geq$ `e`)
  - on trie (**récursivement !**) `s1` et `s2`, on insère `e` entre ces 2 listes triées



Exemple 3 : tri rapide (`tri_rapide (l : α list) : α list`)

Algorithme

- si `l`=[], le résultat est la liste vide ...
- si `l=e::s` alors
  - on partitionne `s` en `s1` (elts de `s` $<$ `e`) et `s2` (elts de `s` $\geq$ `e`)
  - on trie (**récursivement !**) `s1` et `s2`, on insère `e` entre ces 2 listes triées



Partitionner une liste en fonction d'un pivot ?

`partition` : $\alpha \rightarrow \alpha$ `list` $\rightarrow$ (α `list` $*$ α `list`)

(`partition p l`) renvoie (`l1`,`l2`) tel que `l1` contient les élts de `l` strictements inférieurs à `p` et `l2` ceux supérieurs à `p`.

DEMO: tri rapide

Comment s’assurer qu’une fonction récursive est **correcte** ?

Une solution : prouver qu’elle calcule bien le résultat attendu
paramètre de type récursif $\rightarrow$ preuve par récurrence sur sa structure

Comment s'assurer qu'une fonction récursive est **correcte** ?

Une solution : prouver qu'elle calcule bien le résultat attendu
paramètre de type récursif $\rightarrow$ preuve par récurrence sur sa structure

Principe

Pour montrer une propriété P sur une liste `l`

1. Cas de base : montrer que $P([])$ est vrai
2. Récurrence : montrer que $P($`e::s`$)$ est vrai, en suposant $P($`s`$)$ vrai

Comment s'assurer qu'une fonction récursive est **correcte** ?

Une solution : prouver qu'elle calcule bien le résultat attendu
paramètre de type récursif → preuve par récurrence sur sa structure

Principe

Pour montrer une propriété P sur une liste `l`

1. Cas de base : montrer que $P([])$ est vrai
2. Récurrence : montrer que $P(\texttt{e::s})$ est vrai, en suposant $P(\texttt{s})$ vrai

Plus formellement

Pour tout prédicat P sur une liste `l`

$$\left.\begin{array}{l} P(\texttt{[]}) \\ \forall \texttt{e}, \texttt{s}, P(\texttt{s}) \Rightarrow P(\texttt{e::s}) \end{array}\right\} \Rightarrow \forall \texttt{l}, P(\texttt{l})$$

Comment s'assurer qu'une fonction récursive est **correcte** ?

Une solution : prouver qu'elle calcule bien le résultat attendu
paramètre de type récursif → preuve par récurrence sur sa structure

Principe

Pour montrer une propriété P sur une liste `l`

1. Cas de base : montrer que $P([])$ est vrai
2. Récurrence : montrer que $P(\texttt{e::s})$ est vrai, en suposant $P(\texttt{s})$ vrai

Plus formellement

Pour tout prédicat P sur une liste `l`

$$\left.\begin{array}{l} P(\texttt{[]}) \\ \forall \texttt{e}, \texttt{s}, P(\texttt{s}) \Rightarrow P(\texttt{e::s}) \end{array}\right\} \Rightarrow \forall \texttt{l}, P(\texttt{l})$$

Remarques :

- se généralise à tous les types récursifs . . .
- la récurrence sur les entiers est un cas particulier

Exemple de preuve : somme de éléments d'une liste

```
let rec somme (l : int list) : int =
  match l with
    [] → 0
    |e::s → e + (somme s)
```

Exemple de preuve : somme de éléments d’une liste

```
let rec somme (l : int list) : int =
  match l with
    [] → 0
    |e::s → e + (somme s)
```

Prouver que, pour toute liste d’entiers `l` = [`e1`;`e2`; ...; `en`],

$$P(\texttt{l}) \equiv_{def} (\texttt{somme l}) = \Sigma_{i=1}^{n} e_i$$

Exemple de preuve : somme de éléments d'une liste

```
let rec somme (l : int list) : int =
  match l with
    [] → 0
   |e::s → e + (somme s)
```

Prouver que, pour toute liste d'entiers `l` = [`e1`;`e2`; ...; `en`],

$$P(\texttt{l}) \equiv_{def} (\texttt{somme l}) = \Sigma_{i=1}^{n} e_i$$

1. Cas de base (pour la liste vide) :
   (somme []) = 0 et $\Sigma_{i=1}^{i=0}[] = 0$, donc $P([])$ est vrai

Exemple de preuve : somme de éléments d'une liste

```
let rec somme (l : int list) : int =
  match l with
    [] → 0
    |e::s → e + (somme s)
```

Prouver que, pour toute liste d'entiers `l` = [`e1`;`e2`; ...; `en`],

$$P(\texttt{l}) \equiv_{def} (\texttt{somme l}) = \Sigma_{i=1}^{n} e_i$$

1. Cas de base (pour la liste vide) :

   $$(\text{somme } []) = 0 \text{ et } \Sigma_{i=1}^{i=0}[] = 0, \text{ donc } P([]) \text{ est vrai}$$

2. Récurrence structurelle
   - On suppose que pour toute liste `s` = [`e1`;`e2`; ...;`en`] on a $P(\texttt{s})$:

     $$(\texttt{somme s}) = \Sigma_{i=1}^{n}\texttt{ei}$$ (hyp. de récurrence)

Exemple de preuve : somme de éléments d'une liste

```
let rec somme (l : int list) : int =
  match l with
    [] → 0
    |e::s → e + (somme s)
```

Prouver que, pour toute liste d'entiers `l` = [`e1`;`e2`; ...; `en`],

$$P(\texttt{l}) \equiv_{def} (\texttt{somme l}) = \Sigma_{i=1}^{n} e_i$$

1. Cas de base (pour la liste vide) :

   (somme []) = 0 et $\Sigma_{i=1}^{i=0}[] = 0$, donc $P([])$ est vrai

2. Récurrence structurelle

   - On suppose que pour toute liste `s` = [`e1`;`e2`; ...;`en`] on a $P(\texttt{s})$:

     $(\texttt{somme s}) = \Sigma_{i=1}^{n}\texttt{ei}$ (hyp. de récurrence)

   - On a alors, pour la liste `e0::s` = [`e0`;`e1`;`e2`; ... ;`en`] :

     `(somme e0::s) = e0 + (somme s)` (def. de la fonction `somme`)

Exemple de preuve : somme de éléments d'une liste

```
let rec somme (l : int list) : int =
  match l with
   [] → 0
   |e::s → e + (somme s)
```

Prouver que, pour toute liste d'entiers l = [e1;e2; ...; en],

$$P(\texttt{l}) \equiv_{def} (\texttt{somme l}) = \Sigma_{i=1}^{n} e_i$$

1. Cas de base (pour la liste vide) :

$$(\text{somme } []) = 0 \text{ et } \Sigma_{i=1}^{i=0}[] = 0, \text{ donc } P([]) \text{ est vrai}$$

2. Récurrence structurelle
   - On suppose que pour toute liste s = [e1;e2; ...;en] on a $P(\texttt{s})$:

     $$(\texttt{somme s}) = \Sigma_{i=1}^{n}\texttt{ei} \quad \text{(hyp. de récurrence)}$$

   - On a alors, pour la liste e0::s = [e0;e1;e2; ... ;en] :

     $$(\texttt{somme e0::s}) = \texttt{e0 + (somme s)} \quad \text{(def. de la fonction somme)}$$

     or,

     $$\begin{aligned} \texttt{e0 + somme s} &= \texttt{e0} + \Sigma_{i=1}^{n}\texttt{ei} \quad \text{(par hyp. de récurrence)} \\ &= \Sigma_{i=0}^{n}\texttt{ei} \end{aligned}$$

     On en déduit donc que $P(\texttt{e::s})$ est vrai.

Conclusion

Les listes : un type de données **incontournable**

- Peuvent être définies explicitementt par un type union (récursif) :
  - constructeurs `Cons` et `Nil`
  - "first-class citizens"
  - toutes les règles de typage s'appliquent, pattern-matching, etc.
- En pratique : on utilise plutôt le type `list` de OCaml

  ::, [], [`v1`;`v2`;...;`vn`], `@`, ...
- fonctions récursives sur les listes :
  - identifier et définir le(s) cas de base
  - identifier et définir le(s) cas de récurrence
- 2 algorithmes de tri : tri par insertion et tri par sélection
- notion de preuve (par récurrence) de fonctions récursives

Important

- Vérifiez que vous savez écrire les fonctions vues dans ce cours
- Jetez un coup d'oeil aux opérateurs prédéfinis du type `List` ...