Algorithmique - Programmation 1

Cours 5

Université Henri Poincaré

CESS Epinal

Automne 2008

Plan

Rappels : Types en Caml

- Le type d'une donnée nous indique quelles **valeurs** peut prendre cette donnée, et quelles **opérations** peuvent lui être appliquées
- Types de bases en Caml :
  `int, float, bool, char, string`
- Types plus complexes : *combinaisons de types*
  (exemples : produits cartésiens, **listes**)
- NB : si le type d'une donnée ne peut être déduit par le contexte, l'expression est **polymorphe**

Plan

Les listes

- Contrairement aux tuples (produits cartésiens), les listes regroupent **un nombre quelconque** de données **de même type**
- Les listes sont construites
  - au moyen des crochets `[ ]`
  - au moyen de l'opérateur `::`
- Les crochets (`[]`) représentent la **liste vide**
- L'opérateur `::` est appelé constructeur de listes :

```
:: 'a -> 'a list -> 'a list
```

Les listes (suite)

- Exemples de listes en `Caml` :

```
# [1;2;3];;
```

Les listes (suite)

- Exemples de listes en `Caml` :

```
# [1;2;3];;
- :  int list = [1; 2; 3]
```

Les listes (suite)

- Exemples de listes en `Caml` :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
```

Les listes (suite)

- Exemples de listes en `Caml` :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
- :  'a list = []
```

Les listes (suite)

- Exemples de listes en `Caml` :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
- :  'a list = []

# 1 ::  2 ::  [];;
```

Les listes (suite)

- Exemples de listes en Caml :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
- :  'a list = []

# 1 ::  2 ::  [];;
- :  int list = [1; 2]
```

Les listes (suite)

- Exemples de listes en Caml :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
- :  'a list = []

# 1 ::  2 ::  [];;
- :  int list = [1; 2]

# [(function x -> x+1); (function y -> y-1)];;
```

Les listes (suite)

- Exemples de listes en Caml :

```
# [1;2;3];;
- :  int list = [1; 2; 3]

# [];;
- :  'a list = []

# 1 ::  2 ::  [];;
- :  int list = [1; 2]

# [(function x -> x+1); (function y -> y-1)];;
- :  (int -> int) list = [<fun>; <fun>]
```

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
```

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
- :  int list = [1; 2]
```

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
- :  int list = [1; 2]

# [1,2];;
```

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
- :  int list = [1; 2]

# [1,2];;
- :  (int * int) list = [(1, 2)]
```

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
- :  int list = [1; 2]

# [1,2];;
- :  (int * int) list = [(1, 2)]
```

- Attention à bien respecter la règle de typage des listes (tous les éléments doivent être du même type !)

Les listes (suite)

- Les éléments d'une liste sont séparés par des **point-virgules**
- Exemple :

```
# [1;2];;
- :  int list = [1; 2]
```

```
# [1,2];;
- :  (int * int) list = [(1, 2)]
```

- Attention à bien respecter la règle de typage des listes (tous les éléments doivent être du même type !)
- **Concaténation** de listes au moyen de l'opérateur **@**

```
# [1;2] @ [3; 4];;
- :  int list = [1; 2; 3; 4]
```

Les listes (suite)

- Les traitements sur les listes sont souvent **récursifs**

Les listes (suite)

- Les traitements sur les listes sont souvent **récursifs**
- **Schéma** :

```
# let rec ma_fonction =
      function [] -> (* cas d'arrêt *)
           | t::r -> (* cas récursif en fonction
                        de la tête et du reste *) ;;
```

Les listes (suite)

- Les traitements sur les listes sont souvent **récursifs**
- **Schéma** :

```
# let rec ma_fonction =
      function [] -> (* cas d'arrêt *)
           | t::r -> (* cas récursif en fonction
                        de la tête et du reste *) ;;
```

- Exemple : **nombre d'occurrences** d'un paramètre x

```
# let rec nbocc =
     function x ->
     function [] -> 0
          | t::r -> let howmany = (nbocc x r) in
                    if (x = t) then 1 + howmany
                               else howmany ;;
```

Les listes (suite)

- Les traitements sur les listes sont souvent **récursifs**
- **Schéma** :

```
# let rec ma_fonction =
      function [] -> (* cas d'arrêt *)
           | t::r -> (* cas récursif en fonction
                        de la tête et du reste *) ;;
```

- Exemple : **nombre d'occurrences** d'un paramètre x

```
# let rec nbocc =
     function x ->
     function [] -> 0
          | t::r -> let howmany = (nbocc x r) in
                    if (x = t) then 1 + howmany
                               else howmany ;;
val nbocc :  'a -> 'a list -> int = <fun>
```

Les listes (suite)

- Le compilateur `Caml` intègre une bibliothèque de fonctions sur les listes :
  - ```
    # List.mem ;;
    - : 'a -> 'a list -> bool = <fun>
    ```
  - ```
    # List.length ;;
    - : 'a list -> int = <fun>
    ```
  - ```
    # List.hd ;;
    - : 'a list -> 'a = <fun>
    ```
  - ```
    # List.tl ;;
    - : 'a list -> 'a list = <fun>
    ```
  - ```
    # List.rev ;;
    - : 'a list -> 'a list = <fun>
    ```
    ```

Les listes (suite)

- Possibilité d'appliquer un traitement à une liste pour produire une nouvelle liste : **mapping**
- Utilisation de la fonction `List.map` :

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Exemple : **incrémentation** des éléments d'une liste

```
# List.map (function x -> x + 1) [1; 2; 3];;
- : int list = [2; 3; 4]
```

- Définition récursive de la fonction `map` ?

Les listes (suite)

- Traitement sur une liste avec **agrégation** des résultats intermédiaires : **folding**
- Utilisation des fonctions `List.fold_left` et `List.fold_right` :

```
# List.fold_left ;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

- Exemple : **somme des éléments** d'une liste

```
# List.fold_left
    (function x -> function y -> x + y) 0 [1; 2; 3];;
- : int = 6
```

- Affichage des éléments d'une liste séparés par des "," ?

Les listes (suites)

- Comment `Caml` évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
- :  float list = [2.; 4.; 11.]
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
- :  float list = [2.; 4.; 11.]

# let rec gen_n = function 0 -> []
                  | n -> (gen_n (n-1)) @ [n];;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
- :  float list = [2.; 4.; 11.]

# let rec gen_n = function 0 -> []
                  | n -> (gen_n (n-1)) @ [n];;
val gen_n :  int -> int list = <fun>
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
- :  float list = [2.; 4.; 11.]

# let rec gen_n = function 0 -> []
                  | n -> (gen_n (n-1)) @ [n];;
val gen_n :  int -> int list = <fun>

# gen_n 3;;
```

Les listes (suites)

- Comment Caml évalue-t-il les expressions suivantes ?

```
# let a = List.hd [];;
Exception:  Failure "hd".

# List.tl [1; 2; 3];;
- :  int list = [2; 3]

# let c = 1.2 ::  [2;3];;
This expression has type int but is here used with
type float

# List.map sqrt [4.  ; 16.  ; 121.];;
- :  float list = [2.; 4.; 11.]

# let rec gen_n = function 0 -> []
                  | n -> (gen_n (n-1)) @ [n];;
val gen_n :  int -> int list = <fun>

# gen_n 3;;
- :  int list = [1; 2; 3]
```

Plan

Utilisation des listes

- Les listes permettent de modéliser des données en nombre indéfini
- La plupart des compilateurs pour langages fonctionnels (dont `OCaml`) incluent des librairies offrant divers traitements sur les listes (appartenance, longueur, *etc*)
- Nous allons voir 3 exemples d'utilisation des listes :
  - modélisation du calcul des prédicats
  - modélisation des relations binaires
  - modélisation des vecteurs

Calcul des prédicats

- Comment exprimer quelque chose du genre : *"pour tout élément x d'un ensemble E, le prédicat $P(x)$ est vrai"* ?
- Par prédicat, on entend une fonction retournant un booléen

  Exemples :
  - `let pair n =  (n mod 2) = 0;;`
  - `let positif n = (n > 0);;`
  - `let entier n =(int_of_float (float_of_int n))=n;;`
- Notion de quantification : **pour tout** et **il existe**
- En Caml :

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.exists ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
```

Calcul des prédicats (suite)

- Comment écrire une fonction qui retourne vrai si tous les éléments d'une liste d'entiers passée en paramètre sont impairs ?
- Comment réécrire les fonctions `List.for_all` et `List.exists` sans utiliser le mot-clé `rec` ?
- Indices :
  - `for_all` d'une fonction à valeur booléenne `p` sur une liste `[x1; x2 ... xn]` signifie que `(p x1)` **et** `(p x2)` **et** ... `(p xn)` valent vrai
  - `exists` d'une fonction à valeur booléenne `p` sur une liste `[x1; x2 ... xn]` signifie que `(p x1)` **ou** `(p x2)` **ou** ... `(p xn)` vaut vrai
  - on peut réutiliser les fonctions d'agrégation `List.fold_left` et/ou `List.fold_right`

Modélisation d'une relation binaire

- On peut représenter une relation binaire $\mathcal{R}$ par une liste de couples L_c, le sens de cette liste étant le suivant :

$$x\mathcal{R}y \Leftrightarrow (x, y) \in L_c$$

- Exemple : représentation de la relation $<$ (inférieur strict) sur l'ensemble $\{1; 2; 3\}$

  $\longrightarrow$ `[ (1, 2) ; (1, 3) ; (2, 3) ]`

- Comment vérifier qu'une relation binaire $\mathcal{R}$ est une relation d'équivalence ?

  Rappel : une relation d'équivalence se caractérise par le fait qu'elle est **réflexive**, **symétrique** et **transitive**

Modélisation d'une relation binaire (suite)

- **Réflexivité** : $\forall x \in E,\ x\mathcal{R}x$ se traduit en `Caml` par

Modélisation d'une relation binaire (suite)

- **Réflexivité** : $\forall x \in E,\ x\mathcal{R}x$ se traduit en `Caml` par

```
# let reflexive = function l ->
          List.for_all
              (function x -> List.mem (x,x) l)
              (flat l);;
```

Modélisation d'une relation binaire (suite)

- **Réflexivité** : $\forall x \in E,\ x\mathcal{R}x$ se traduit en `Caml` par

```
# let reflexive = function l ->
          List.for_all
              (function x -> List.mem (x,x) l)
              (flat l);;
```

où `flat` est une fonction d'applatissement :

```
# let rec flat = function [] -> []
              | (a,b) ::  r -> [a;b] @ (flat r);;
```

Modélisation d'une relation binaire (suite)

- **Réflexivité** : $\forall x \in E,\ x\mathcal{R}x$ se traduit en `Caml` par

```
# let reflexive = function l ->
          List.for_all
              (function x -> List.mem (x,x) l)
              (flat l);;
```

où `flat` est une fonction d'applatissement :

```
# let rec flat = function [] -> []
                | (a,b) ::  r -> [a;b] @ (flat r);;
```

- Question : qu'en est-il des doublons dans la liste applatie ?

Modélisation d'une relation binaire (suite)

- **Symétrie** : $\forall x, y \in E,\ x\mathcal{R}y \Rightarrow y\mathcal{R}x$

Modélisation d'une relation binaire (suite)

- **Symétrie** : $\forall x, y \in E,\ x\mathcal{R}y \Rightarrow y\mathcal{R}x$

```
# let symetrique = function l ->
                List.for_all
                    (function a,b -> List.mem (b,a) l)
                    l;;
```

Modélisation d'une relation binaire (suite)

- **Symétrie** : $\forall x, y \in E,\ x \mathcal{R} y \Rightarrow y \mathcal{R} x$

```
# let symetrique = function l ->
                List.for_all
                    (function a,b -> List.mem (b,a) l)
                    l;;
```

- **Transitivité** : $\forall x, y, z \in E,\ x \mathcal{R} y \wedge y \mathcal{R} z \Rightarrow x \mathcal{R} z$

Modélisation d'une relation binaire (suite)

- **Symétrie** : $\forall x, y \in E,\ x\mathcal{R}y \Rightarrow y\mathcal{R}x$

```
# let symetrique = function l ->
                List.for_all
                    (function a,b -> List.mem (b,a) l)
                    l;;
```

- **Transitivité** : $\forall x, y, z \in E,\ x\mathcal{R}y \wedge y\mathcal{R}z \Rightarrow x\mathcal{R}z$

```
# let rec trans = function l -> function x,y ->
  function [] -> true
  | (a,b) ::  r -> if (a=y) then (List.mem (x,b) l)
                              else (trans l (x,y) r);;
```

Modélisation d'une relation binaire (suite)

- **Symétrie** : $\forall x, y \in E,\ x\mathcal{R}y \Rightarrow y\mathcal{R}x$

```
# let symetrique = function l ->
            List.for_all
               (function a,b -> List.mem (b,a) l)
               l;;
```

- **Transitivité** : $\forall x, y, z \in E,\ x\mathcal{R}y \wedge y\mathcal{R}z \Rightarrow x\mathcal{R}z$

```
# let rec trans = function l -> function x,y ->
  function [] -> true
  | (a,b) ::  r -> if (a=y) then (List.mem (x,b) l)
                            else (trans l (x,y) r);;

# let transitive = function l ->
      List.for_all
          (function a,b -> trans l (a,b) l)
          l;;
```

Modélisation des vecteurs

- Représentation d'un vecteur de $\mathbb{R}^n$ au moyen d'une liste de réels, exemple : `[1.2 ; 2.5 ; 5.6]` $\in \mathbb{R}^3$
- Comment modéliser la multiplication d'un vecteur par un scalaire ? Ecrire la fonction `mult_scal`
- Comment calculer la norme d'un vecteur de $\mathbb{R}^n$? Ecrie la fonction `norme` de 2 façons différentes (sans et avec `fold_left`)
- Comment calculer le produit scalaire de 2 vecteurs ? Ecrire la fonction `prod_scal`

Conclusion

- Intérêt des listes : représentation d'un nombre indéfini de données **de même type**
- Notation `Caml` : **[ ]** ou **::** pour construire une liste, et les éléments sont séparés par des **;**
- **Fonctions prédéfinies** pour le traitement des listes regroupées dans le module `List`
- **Schéma** de traitement **récursif** d'une liste :

```
# let rec ma_fonction =
    function [] -> (* cas d'arrêt *)
         | t::r -> (* cas récursif en fonction
                      de la tête et du reste *) ;;
```