Lycée Louis-le-Grand
Année 2003–2004

# Rappels de Caml

option informatique





## 1 Nombres de Fibonacci

On utilisera ici la définition suivante de la suite $(F_n)$ des nombres de Fibonacci :

$$F_0 = 0, F_1 = 1, \quad \forall n \geqslant 2, F_n = F_{n-1} + F_{n-2}.$$

La version naïve consiste à écrire :

```
let rec fibonacci_naif n =
  if n = 0 then 0
  else if n = 1 then 1
  else (fibonacci_naif (n-1)) + (fibonacci_naif (n-2)) ;;
```

- Pourquoi est-ce naïf ?
- Quel est le coût à l'exécution de l'appel `fibonacci_naif n` ?



MCours.com



En fait, le calcul de $F_n$ ne nécessite que les valeurs des deux précédents termes de la suite. C'est ce qui donne l'idée de la version habituelle :

```
let fibonacci n =
  let rec fib_aux n a b =
    if n = 0 then a
    else fib_aux (n-1) b (a + b)
  in
    fib_aux n 0 1 ;;
```

- On peut aussi écrire `let fibonacci = function n ->`
- Quel est le coût à l'exécution de l'appel `fibonacci n` ?
- Comment prouver la validité de notre fonction ?





Les fanatiques inconditionnels du mode impératif auront peut-être réussi à écrire :

```
let fibonacci_iteratif n =
  let a = ref 0 and b = ref 1 in
    if n = 0 then !a
    else
      begin
       for i = 1 to n do
         let b' = !a + !b in
           a := !b ;
           b := b'
       done ;
       !a
      end ;;
```

- Utilisation des références
- Invariant de boucle ?
- Comparaison. . .

Notant $X_n = \begin{pmatrix} F_n \\ F_{n+1} \end{pmatrix}$, la récurrence de Fibonacci se traduit matriciellement par $X_{n+1} = F \times X_n$ où $F = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$.

Alors $X_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$ et $X_n = F^n \times X_0$.

Le calcul se ramène donc à celui des puissances d'une matrice fixée.
Or il existe une méthode astucieuse (*id est* peu coûteuse) pour évaluer une puissance.
On observe que $F^n = \begin{cases} (F^p)^2, & \text{si } n = 2p; \\ F \times (F^p)^2, & \text{si } n = 2p+1; \end{cases}$ et, par exemple, on peut écrire l'exponentiation entière de la façon suivante :

```
let rec exponentiation x n =
  if n = 0 then 1
  else let y = exponentiation x (n/2) in
    if n mod 2 = 0 then y * y else x * y * y ;;
```

- ▷ Quel est le coût à l'exécution de l'appel `exponentiation x n` ?
- ▷ Comment prouver la validité de notre fonction ?





On déduit des remarques précédentes la version matricielle du calcul des nombres de Fibonacci.

```
let produit_matriciel (a,b,c,d) (a',b',c',d') =
  (a*a'+b*c',a*b'+b*d',c*a'+d*c',c*b'+d*d') ;;

let rec puissance m n =
  if n = 0 then (1,0,0,1)
  else if n = 1 then m
  else let p = puissance m (n/2) in
    if n mod 2 = 0 then produit_matriciel p p
    else produit_matriciel m (produit_matriciel p p) ;;

let fibonacci_matriciel n =
  let f = (0,1,1,1) in
    match puissance f n with (_,a,_,_) -> a ;;
```

- ▷ Choix de représenter une matrice par un quadruplet
- ▷ Expression `match ... with ...` et utilisation de `_`





Les mathématiciens savent résoudre les récurrences du type de celle de Fibonacci.
Ils nous apprennent que

$$\forall n \in \mathbb{N}, F_n = \frac{\varphi^n - \bar{\varphi}^n}{\sqrt{5}}$$

où $\varphi = \frac{1+\sqrt{5}}{2}$ et $\bar{\varphi} = \frac{1-\sqrt{5}}{2}$.
Ceci permet un calcul en temps constant de $F_n$ :

```
let arrondi x =
  int_of_float (floor (x +. 0.5)) ;;

let fibonacci_flottant n =
  arrondi (((1.0 +. sqrt(5.0))/. 2.0) **. (float_of_int n) /. sqrt(5.0)) ;;
```

- ▷ Questions de parenthésage
- ▷ Problème de la précision des calculs
- ▷ Problème des grands nombres





## 2 Vecteurs et quick-sort

On s'intéresse au problème du tri en ordre croissant des éléments d'un vecteur d'entiers supposés deux à deux distincts.
On souhaite réaliser un tri *en place*, c'est-à-dire réorganiser les éléments du vecteur, sans introduire de nouveau vecteur.
L'idée du tri-rapide (*quick-sort* en anglais) est la suivante : on choisit un élément, appelé pivot, et on réorganise les éléments du vecteur $v$ de sorte que si $i_0$ est l'indice où figure le pivot, $v_i$ est inférieur au pivot si $i < i_O$ et $v_i$ est supérieur au pivot si $i > i_0$.
Il suffit alors d'appliquer récursivement le tri-rapide aux deux tranches du vecteur ainsi constituées.

On obtient facilement la fonction suivante :

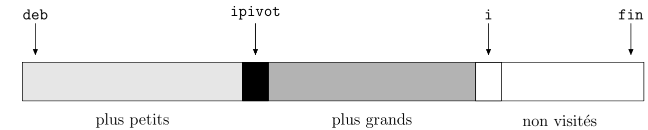
```
let quick_sort v =
  let rec quick_sort_aux v deb fin =
    if deb < fin then
      let i = partage v deb fin in
       (quick_sort_aux v deb (i-1) ; quick_sort_aux v (i+1) fin )
  in
    quick_sort_aux v 0 (vect_length v - 1) ;
    v ;;
```

On a écrit une fonction récursive `quick_sort_aux` qui travaille sur une tranche $\{v_i, \texttt{deb} \leqslant i \leqslant \texttt{fin}\}$.
Toute la difficulté réside dans l'écriture de la fonction de partage.

- ▷ Indexation des vecteurs
- ▷ Parenthèses ou `begin` ... `end`
- ▷ Sémantique du `;`








```
let partage v deb fin =
  let pivot = v.(deb) and ipivot = ref deb in
    for i = deb + 1 to fin do if v.(i) < pivot then begin
      v.(!ipivot) <- v.(i) ;
      incr ipivot ;
      v.(i) <- v.(!ipivot)
    end done ;
    v.(!ipivot) <- pivot ;
    !ipivot ;;
```

- ▷ On choisit le premier élément comme pivot
- ▷ La case du pivot n'est remplie qu'à la fin
- ▷ Invariant de boucle



MCours.com



## 3 Manipulation des listes

Rappelons quelques fonctions élémentaires.

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

let rec dernier = function
  | [] -> failwith "liste vide"
  | [ a ] -> a
  | _ :: q -> dernier q ;;

let miroir l =
  let rec aux m = function
    | [] -> m
    | t :: q -> aux (t :: m) q
  in
    aux [] l ;;
```





Parmi les fonctions précédentes, `dernier` est récursive terminale, mais pas `longueur`.
Il n'est pas difficile d'écrire une version récursive terminale de `longueur` :

```
let longueur_terminal l =
  let rec aux n = function
    | [] -> n
    | _ :: q -> aux (n+1) q
  in
    aux 0 l ;;
```

- ▷ Intérêt de la récursivité terminale
- ▷ Toujours préférer la version la plus naturelle, la plus facile à lire, et à maintenir

## Fonctionnelles standard sur les listes

```
let rec map f = function
  | [] -> []
  | t :: q -> (f t) :: (map f q) ;;

let rec it_list f x = function
  | [] -> x
  | t :: q -> it_list f (f x t) q ;;

let rec list_it f l x = match  l with
  | [] -> x
  | t :: q -> f t (list_it f q x) ;;
```

- ▷ Utilisation des fonctionnelles
- ▷ Écriture des versions récursives terminales





```
let map_terminal f l =
  let rec aux m = function
    | [] -> miroir m
    | t :: q -> aux ((f t) :: m) q
  in
    aux [] l ;;

let list_it_terminal f l x =
  let rec aux y = function
    | [] -> y
    | t :: q -> aux (f t y) q
  in
    aux x (miroir l) ;;
```





Pour illustrer l'utilisation des *clauses gardées*, on se propose d'écrire la suppression d'un objet dans une liste (ou bien une fois, ou bien à chaque fois).

```
let rec del1 x = function
  | [] -> []
  | t :: q when t = x -> q
  | t :: q -> t :: (del1 x q) ;;

let rec del_all x = function
  | [] -> []
  | t :: q when t = x -> del_all x q
  | t :: q -> t :: (del_all x q) ;;
```

- ▷ La clause gardée peut être remplacée par une conditionnelle
- ▷ N'utiliser les clauses gardées que si le code devient plus explicite





## Application : le tri-fusion

Le tri-fusion (*merge-sort* en anglais) est un tri qui porte cette fois sur les listes. Une nouvelle liste, triée, est construite. Ce n'est pas un tri en place. Il consiste tout simplement en trois étapes :

1. partager la liste en deux sous-listes de tailles égales (à une unité près) ;
2. trier récursivement chacune des deux sous-listes ;
3. fusionner les deux listes triées.

- ▷ Caml se préoccupe tout seul de la gestion de la mémoire : il le fait très bien !
- ▷ L'algorithme est en $O(n \lg n)$ : c'est optimal, dans tous les cas.

```
let rec découpe = function
  | [] -> [], []
  | [ a ] -> [a], []
  | a :: b :: q -> let l,m = découpe q in (a :: l), (b :: m) ;;

let rec fusionne l m = match (l,m) with
  | ([],m) -> m
  | (l,[]) -> l
  | (t :: q, t' :: q') ->
      if t < t' then t :: (fusionne q m)
      else t' :: (fusionne l q') ;;

let rec tri_fusion = function
  | [] -> []
  | [ x ] -> [ x ]
  | l -> let (a,b) = découpe l in fusionne (tri_fusion a) (tri_fusion b) ;;
```





## Listes associatives

Une liste associative est une liste de couples $(c, v)$ où $c$ est un sélecteur et où $v$ est une valeur.
L'opération fondamentale consiste à chercher la valeur associée à un sélecteur donné dans une liste associative. On prend toujours la première valeur trouvée.

```
let rec associe x = function
  | [] -> failwith "introuvable"
  | (a,b) :: _ when a = x -> b
  | _ :: q -> associe x q ;;
```

- Cette fonction est dans la bibliothèque standard de Caml sous le nom `assoc`
- Pour gérer des listes de taille importante, on utilise souvent des tables de hachage





## 4 Types sommes

Nous considérerons ici l'exemple des propositions logiques, avec deux constantes, un opérateur unaire (la négation), deux opérateurs binaires (conjonction et disjonction), et la possibilité d'utiliser des variables.
Cela mène à la définition suivante :

```
type proposition =
  | Vrai
  | Faux
  | Neg of proposition
  | Ou of proposition * proposition
  | Et of proposition * proposition
  | Var of string ;;
```

- Vocabulaire : constructeurs
- Il serait facile d'ajouter l'implication, l'équivalence, etc
- On pourrait faire de même avec des expressions arithmétiques
- On en reparlera dans le cours sur les arbres : il est à noter qu'il n'y a pas de problème de parenthésage





L'expression logique $(\neg p) \wedge (p \vee \neg q)$ correspond à la valeur Caml de type `proposition` suivante :
`Et(Neg(Var "p"),Ou(Var "p",Neg(Var "q")))`.
L'évaluation n'a de sens que dans un contexte, c'est-à-dire une valuation des variables, que nous représenterons ici par une liste associative.

```
let rec evalue contexte = function
  | Vrai -> true
  | Faux -> false
  | Neg p -> not (evalue contexte p)
  | Ou(p,q) -> (evalue contexte p) || (evalue contexte q)
  | Et(p,q) -> (evalue contexte p) && (evalue contexte q)
  | Var v -> associe v contexte ;;
```

On peut remarquer que les listes Caml habituelles forment aussi un type somme qu'on pourrait écrire :

```
type 'a liste = Vide | L of 'a * 'a liste ;;
```

Le filtrage standard s'écrirait alors :

```
let miroir l =
  let rec aux m = function
     | Vide -> m
     | L(t,q) -> aux L(t,m) q
  in aux Vide l ;;
```





## 5 Types produits

Nous considérerons ici un typage des graphes non dirigés ; chaque sommet sera coloré, pour des raisons qui apparaîtront clairement plus tard.

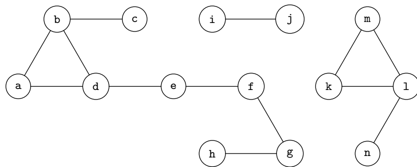
```
type couleur = Bleu | Blanc ;;

type 'a sommet = { etiq : 'a ;
        mutable voisins : 'a sommet list ;
              mutable c : couleur } ;;
```

- ▷ Vocabulaire : enregistrement, champ, sélecteur
- ▷ Comparaison avec les $p$-uplets (ordre des champs indifférent)
- ▷ Intérêt/inconvénient des champs mutables



MCours.com






```
let a = { etiq = 'a' ; voisins = [] ; c = Blanc }
and b = { etiq = 'b' ; voisins = [] ; c = Blanc }
...
and n = { etiq = 'n' ; voisins = [] ; c = Blanc } ;;

a.voisins <- [b;d] ; b.voisins <- [a;c;d] ; c.voisins <- [b] ;
...
let G = [a;b;c;d;e;f;g;h;i;j;k;l;m;n] ;;
```

- ▷ Comment faire sans champ mutable ?





On cherche à calculer la composante connexe d'un sommet du graphe. L'idée est de colorer les sommets validés au fur et à mesure, afin de ne pas parcourir les cycles éventuels indéfiniment.

```
let composante_connexe s =
  let rec itere acceptes = function
    | [] -> acceptes
    | t :: q when t.c = Bleu -> itere acceptes q
    | t :: q -> t.c <- Bleu ; itere (t :: acceptes) (t.voisins @ q)
  in
    s.c <- Bleu ;
    let res = itere [ s ] s.voisins in
      do_list (function t -> t.c <- Blanc) res ;
      res ;;
```

- ▷ Il ne faut pas oublier de colorer le sommet de départ avant de lancer l'itération
- ▷ On recolorie en blanc après avoir trouvé le résultat
- ▷ Comment prouver la terminaison du programme ? sa validité ?
- ▷ L'itération se passe-t-elle différemment si on écrit `(q @ t.voisins)` plutôt que `(t.voisins @ q)` ?