Cours de l'option informatique

Lycée Louis-le-Grand

Année 2000–2001

Rappels sur le langage Caml

sommaire

— liaisons globales et locales ;

— fonctions sur les listes ;

— itérateurs sur les listes ;

— types et filtrages ;

— style impératif ;

— effets de bord

liaisons globales et locales

```
#let a = 2 ;;
a : int = 2

#a + 3 ;;
- : int = 5

#let b = 3 in a + b ;;
- : int = 5

#b ;;
Entrée interactive:
>b ;;
>^
L'identificateur b n'est pas défini.
```

types Caml à utiliser

types simples `int float bool char unit`

types construits de base `list vect string`
attention : les deux derniers sont mutables !

types somme Ils peuvent être récursifs ! Un exemple plus simple :
```
type carte = As | Roi | Dame | Valet | N of int
```

types produit Ce sont les enregistrements d'autres langages :
```
type citoyen = { nom : string ; no_sécu : int ; ... }
```

fonctions sur les listes

```
longueur : 'a list -> int
accolle : 'a list -> 'a list -> 'a list
membre : 'a -> 'a list -> bool
adjoint : 'a -> 'a list -> 'a list
union : 'a list -> 'a list -> 'a list
intersection : 'a list -> 'a list -> 'a list
miroir : 'a list -> 'a list
produit_cartésien :
          'a list -> 'a list -> ('a * 'a) list
```

Instruction `failwith "message d'erreur"`

```
let rec longueur = function
    | [] -> 0
    | _ :: q -> longueur q + 1 ;;
let rec accole a b = match a with
    | [] -> b
    | t :: q -> t :: (accole q b) ;;
let rec membre x = function
    | [] -> false
    | t :: q -> t = x || membre x q ;;
let rec adjoint x = function
    | [] -> [ x ]
    | t :: q -> if t = x then t :: q else t :: (adjoint x q) ;;
let rec union a b = match a with
    | [] -> b
    | t :: q -> union q (adjoint t b) ;;
let miroir a =
    let rec déverse a b = match a with
        | [] -> b
        | t :: q -> déverse q (t :: b)
    in
    déverse a [] ;;
```

itérateurs sur les listes

```
map : ('a -> 'b) -> 'a list -> 'b list
do_list : ('a -> unit) -> 'a list -> unit
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

`it_list f x [ a ; b ; c ]` s'évalue en `f (f (f x a) b) c`

`list_it f [ a ; b ; c ] x` s'évalue en `f a (f b (f c x))`

Applications : écrire les fonctions

```
min_list max_list sum_list flat_list
```

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

let min_list l = it_list min max_int l ;;
(* max_int est le plus grand entier machine donc le neutre de min *)

let sum_list l = it_list (fun a b -> a + b) 0 l ;;

let flat_list l = it_list (fun a b -> a @ b) [] l ;;
```

filtrage standard sur les listes

```
let rec ma_fonction ... l = match l with
    | [] -> ...
    | tête :: queue -> ... (ma_fonction ... queue)
```

ou encore, ce qui revient exactement au même :

```
let rec ma_fonction ... = function
    | [] -> ...
    | tête :: queue -> ... (ma_fonction ... queue)
```

Donc l'utilisation de `function` est en réalité un filtrage.

On n'utilisera `fun` (qui permet de multiples arguments) qu'avec parcimonie.

vecteurs et chaînes

Si `v` est du type `'a vect`, ses éléments sont du type `'a`. On les indexe à partir de 0 : `v.(i)` est l'élément d'indice `i`, qu'on peut modifier **en place** par une instruction `v.(i) <- x`.

Une chaîne de caractères est presque un vecteur de `char` : le caractère d'indice `i` de la chaîne `s` est `s.[i]` qu'on peut modifier comme ci-dessus. L'opérateur de concaténation est noté `^` .

Utiles : `vect_length` `string_length` `init_vect` `make_vect`

```
init_vect : int -> (int -> 'a) -> 'a vect
make_vect : int -> 'a -> 'a vect
```

Attention au piège ! créer une matrice...

types somme

```
type carte = As | Roi | Dame | Valet | N of int ;;
type couleur = Pique | Cœur | Carreau | Trèfle ;;

let a = As and b = N 8 ;;

type carte_à_jouer == carte * couleur ;;
let c = As,Trèfle ;;

let points_belote atout = function
    | As,_ -> 11    | Roi,_ -> 4    | Dame,_ -> 3
    | Valet,c -> if c = atout then 20 else 2
    | (N 9),c when c = atout -> 14
    | (N 10),_ -> 10    | _ -> 0 ;;
```

types produit

Définition du type :

```
type point = { x : int ; y : int ;
               couleur : color ; visible : bool } ;;
```

Création d'une valeur : **tous** les champs doivent être renseignés

```
let a = { visible = true ; couleur = red ; x = 0 ; y = 3 }
```

Filtrage : on peut ne filtrer que sur **quelques** champs

```
match x with { visible = true ; couleur = red } -> ...
```

Exemple : les listes revisitées

```
type 'a cellule = { valeur : 'a ; suite : 'a liste }
and  'a liste   = Nil | L of 'a cellule ;;
```

Écrire : `filtrer : ('a -> bool) -> 'a liste -> 'a liste`

On s'inspire du programme habituel :

```
let rec filtrer pred = function
    | [] -> []
    | t :: q -> if pred t then t :: (filtrer pred q)
                else filtrer pred q ;;
```

pour écrire :

```
let rec filtrer pred = function
    | Nil -> Nil
    | L { valeur = t ; suite = q }
        -> if pred t then L { valeur = t ;
                              suite = filtrer pred q }
           else filtrer pred q ;;
```

Récapitulatif des motifs de filtrage

```
<identificateur>    <constante>    _    (<motif>)

<motif sans liaison> | <motif sans liaison>

<motif> as <identificateur>

<motif> when <expression booléenne>

<motif>,<motif>    <motif> :: <motif>

<Constructeur>(<motif>)

{ <sélecteur> = <motif>; ... }
```

Structures de contrôle

```
for <identificateur> = <valeur> (to|downto) <valeur>
do
      <instruction>
done ; ;

while <expression booléenne>
do
      <instruction>
done ; ;
```

Ces deux instructions ont, bien entendu, une valeur, qui est `()` (du type `unit`).

Références

Création : `let a = ref [] and b = ref 2 ;;`

Consulation : `match !a with [] -> 2 + !b`

Modification : `a := 2 :: !a ; b := 3 + !b ;;`

À rapprocher de l'utilisation des vecteurs et des chaînes de caractères.

Champs mutables dans un type produit.
Intérêts et précautions à prendre.

Exemple : quicksort.

Tris classiques

Le tri par insertion

C'est le tri des joueurs des cartes : au fur et à mesure qu'une carte est distribuée, on la range à sa place dans sa main.

Algorithmiquement, il s'agit donc d'écrire une fonction d'insertion dans une liste triée.

Ce tri est quadratique, et adapté à la structure de liste.

Le tri par sélection

On extrait le minimum, on trie récursivement ce qui reste de l'ensemble initial. Il n'y a plus qu'à placer en tête le minimum calculé.

Il s'agit là encore d'un tri quadratique, bien adapté aux listes.

Tri d'une liste par insertion

```
(* tri par insertion *)
let rec insère x = function
    | [] -> [ x ]
    | t :: q -> if x <= t then x :: t :: q
                else t :: (insère x q) ;;

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

Tri d'un tableau par insertion.

- pour chaque i, insérer l'élément $a[i]$ parmi $a[1], \ldots a[i-1]$

où la procédure d'insertion peut être définie par

- prendre le premier j,
  - en descendant à partir de i,
  - tel que $a[j-1] \leq a[i]$,
  - en décalant au fur et à mesure les éléments rencontrés vers la droite
- insérer $a[i]$ à la j-ième place.

```
tri_insertion a =
     let N = (vect_length a)-2 in
        for i = 2 to N do
           let v = a.(i) and j = ref i in
              while (!j>1) & (a.(!j-1)>v) do
                 a.(!j) <- a.(!j-1);
                 j := !j -1
              done;
              a.(!j)<- v
      done;;
```

Tri d'une liste par sélection

```
(* tri par sélection *)
let rec extrait_minimum = function
    | [] -> failwith "Liste vide"
    | [ x ] -> x,[]
    | t :: q -> let m,q' = extrait_minimum q
                in
                if t <= m then t,q else m,t::q' ;;

let rec tri_sélection = function
    | [] -> []
    | l -> let m,r = extrait_minimum l
           in
           m :: (tri_sélection r) ;;
```

Tri d'un tableau par sélection

- on cherche l'élément le plus petit dans le tableau et on l'échange avec le premier élément du tableau

- on cherche l'élément immédiatement supérieur et on l'échange avec le deuxième élément du tableau

- ainsi de suite jusqu'à épuisement du tableau.

Ce tri procède donc par sélection du plus petit élément parmi $a[i], \ldots, a[N]$ puis échange ce plus petit élément avec $a[i]$.

```
let tri_selection a =
     let N = (vect_length a)-2 and min = ref 0 in
        for i = 1 to N-1 do
            min := i;
            for j = i+1 to N do
              if a.(j)<a.(!min) then min := j
            done;
            echange a i !min
          done;;
tri_selection : int vect -> unit = <fun>
```

Le tri-fusion (merge-sort)

On découpe l'ensemble de clés à trier en deux parties à peu près égales, qu'on trie récursivement. Il n'y a plus qu'à fusionner deux ensembles triés.

Ce tri est adapté à la structure de liste, il découle directement du paradigme "diviser pour régner" et s'avère efficace, puisqu'en $O(n \lg n)$.

```
let rec fusion l1 l2 = match l1,l2 with
    | [],_ -> l2
    | _,[] -> l1
    | t1::q1,t2::q2
          -> if t1 <= t2 then t1 :: (fusion q1 l2)
                         else t2 :: (fusion l1 q2) ;;

let rec découpe = function
    | [] -> [],[]
    | [ x ] -> [ x ],[]
    | a::b::q -> let q1,q2 = découpe q
                 in
                 a :: q1,b :: q2 ;;

let rec tri_fusion = function
    | [] -> []
    | [ x ] -> [ x ]
    | l -> let l1,l2 = découpe l
           in
           fusion (tri_fusion l1) (tri_fusion l2) ;;
```

Le tri rapide (quicksort)

Pour trier les éléments d'indices i à j (inclus) d'un tableau v :

1. on pose $x = v_i$;
2. on réorganise le tableau (entre i et j) et on trouve r tel que

$$\begin{aligned} \forall k \in \{i, \ldots, r-1\}, v_k &\leqslant x \\ v_r &= x \\ \forall k \in \{r+1, \ldots, j\}, v_k &> x \end{aligned}$$

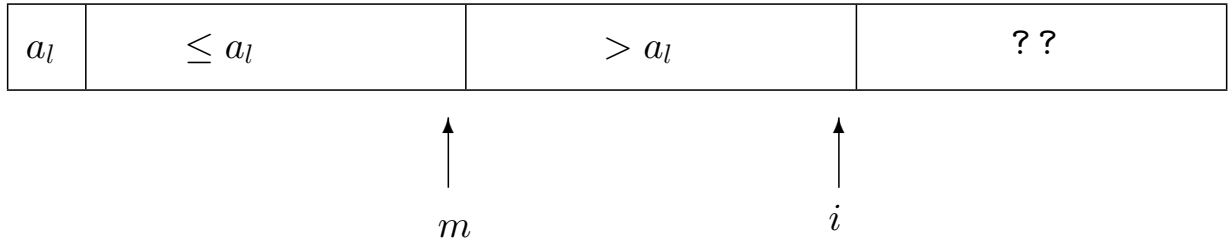
3. on trie (récursivement) les tranches d'indices i à $r-1$ et $r+1$ à j.

Le coût dans le cas le pire (lequel est-il) est $O(n^2)$.

Le coût moyen est $O(n \lg n)$.

Tri rapide

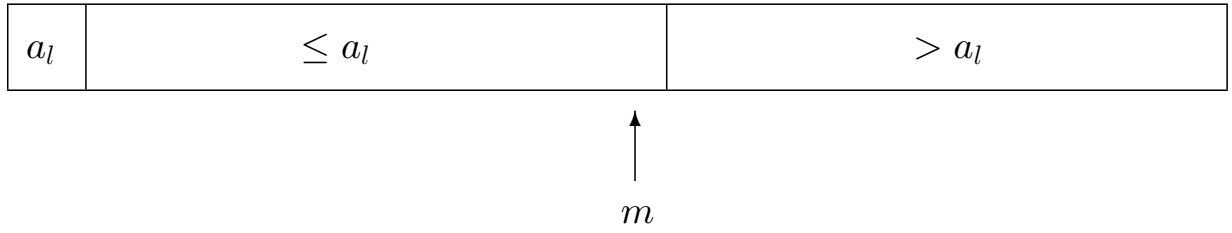
Partition d'un sous-tableau $a_l, \ldots, a_u$:
- boucle indexée par $i \in [l+1, u]$
- avec un deuxième indice m
- de manière à maintenir l'invariant suivant



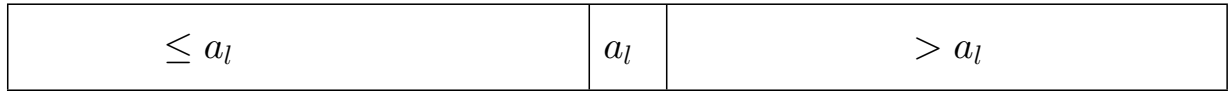
Initialisation : $i = m = l$.

On augmente i de 1, deux cas à considérer
- soit $a_i > a_l$ et dans ce cas il n'y a rien à faire pour conserver l'invariant
- soit $a_i \leq a_l$; il suffit alors d'échanger $a_{m+1} > a_l$ et $a_i \leq a_l$ et d'augmenter m de 1, et l'invariant sera conservé

Après exécution de la totalité de la boucle, on aura donc la situation



Il suffit alors d'échanger a_l et a_m pour avoir la situation



```
let tri_rapide a =
   let N = vect_length a and m = ref 0 and
      echange i j =
       let temp = a.(i) in
          a.(i)<- a.(j);
          a.(j)<- temp
   in
     let partition l u =
       m := l;
       let pivot = a.(l) in
       for i = l+1 to u do
         if a.(i) <= pivot then
           begin
             echange i (!m+1);
             m := !m+1
           end
       done;
       echange l !m
     in
       let rec tri l u =
          if u>l then
            begin
```

```
      partition l u;
      tri l (!m-1);
      tri (!m+1) u
    end
in tri 0 (N-1);;
```

Tri rapide d'une liste

```
let rec partitionne x = function
     | [] -> [],[]
     | t::r -> let (g,d) = partitionne x r in
                  if t<=x then (t::g,d) else (g,t::d);;

let rec tri_rapide = function
    | [] -> []
    | t::r -> let (g,d) = partitionne t r in
              (tri_rapide g)@(t::(tri_rapide d));;
```

Qu'appelle-t-on effet de bord ?
et programmation fonctionnelle ?

Références, vecteurs, champs mutables...

Mais aussi : entrée-sorties, toute gestion de l'interface (bibliothèque `graphics`) et des communications...

(* Piles LIFO et FIFO à l'aide de listes circulaires *)

```
type 'a node = {info:'a; mutable suivant: 'a node}
and 'a liste_circulaire = Vide | Dernier_noeud of 'a node;;

exception Pile_vide;;

(* renvoie les trois méthodes de manipulation d'une pile LFIFO , pile Last or First
In First Out, sous la forme d'un triplet de fonctions:
   empilement en dernier, empilement en premier, depilement du premier   *)
let lfifo () =
         let pile_circ = ref Vide in
         let empile_dernier x =
             match !pile_circ with
                 | Vide -> let rec noeud = {info = x; suivant = noeud} in
                               pile_circ := Dernier_noeud noeud
                 | Dernier_noeud n ->
                           let nouveau_noeud = {info = x; suivant = n.suivant} in
                           n.suivant <- nouveau_noeud;
                           pile_circ := Dernier_noeud nouveau_noeud
         and empile_premier x =
             match !pile_circ with
                 | Vide -> let rec noeud = {info = x; suivant = noeud} in
                               pile_circ := Dernier_noeud noeud
                 | Dernier_noeud n ->
                           let nouveau_noeud = {info=x; suivant = n.suivant} in
                           n.suivant <- nouveau_noeud
```

```
        and depile_premier () =
           match !pile_circ with
                | Vide -> raise Pile_vide
                | Dernier_noeud n ->
                            if n.suivant = n (* un seul element dans la pile *)
                                then
                                   begin
                                      pile_circ:=Vide;
                                      n.info
                                    end
                                else (* au moins deux elements dans la pile *)
                                   begin
                                      let x = n.suivant.info in
                                      n.suivant <- n.suivant.suivant;
                                      x
                                   end
        in (empile_dernier, empile_premier, depile_premier);;

(* renvoie les deux méthodes de manipulation d'une pile FIFO, empilement et depilement *)
let fifo () = let (push,_,pop) = lfifo () in (push,pop);;

(* renvoie les deux méthodes de manipulation d'une pile LIFO, empilement et depilement *)
let lifo () = let (_,push,pop) = lfifo () in (push,pop);;
```