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.
```


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.

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 ; ... }`

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.

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 $O(n \lg n)$.


```
let quicksort v =
    let n = vect_length v
    in
    let scinde i j =
        let r = ref i
        and x = v.(i)
        in      (* v.(i .. r-1) <= x = v.(r) < v.(r+1 .. k) *)
        for k = i + 1 to j do if v.(k) <= x then
              ( v.(!r) <- v.(k) ; incr r ; v.(k) <- v.(!r) )
        done ;
        v.(!r) <- x ;
        !r
    in
    let rec tri i j = if i < j then
        let k = scinde i j
        in ( tri i (k - 1) ; tri (k + 1) j )
    in
    tri 0 (n - 1) ;;
```


Autres 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.

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 $O(n \lg n)$.

```
(* 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 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) ;;
```



```
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) ;;
```


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...