Programmation Fonctionnelle
Cours 3
Listes, filtrage par motif

October 2, 2013

Le type des listes

- Type prédéfini de OCaml
- En fait, il n'y a pas un seul type liste, mais il y a des listes d'entiers, de flottants, de booléens, etc.
- On dit que `list` est un type polymorphe : pour tout type *t*, il y a un type *t* `list`.
- On peut former des listes d'éléments de n'importe quel type (fonctions ou listes incluses), mais :
- Les listes sont homogènes : tous les éléments de la même liste doivent être du même type.

Exemples (lists1.ml)

```
(* definir par enumeration des elements *)
(* entre [ et ], separes par ;           *)
[1; 2; 3];;
[4.0; 7e2];;
[42; 2.0; "toto"];; (* erreur de typage *)
["ab"; "cd"];;
[sin; cos];;
[int_of_float; float_of_int];; (* erreur typage *)
[1;2] = [3;4];;
[1;2] = [1.0; 2.0];; (* erreur de typage *)
[1, 2, 3];; (* pas confondre ; et , *)
```


Variable de type

- Les listes sont homogènes : tous les éléments doivent avoir le même type (qui est donc utilisé pour déterminer le type de liste)
- Quel est le le type de la liste vide ?
- Il s'agit d'un type polymorphe `' list.'` est une variable de type. Les variables de type commencent sur '.
- Les variables de types `' a`, `'b`, `'c` se pronocent *alpha*, *beta*, *gamma*.

Exemples (lists2.ml)

```
[];; (* liste vide *)

[] < [1; 2];;

[] < ["hello"; "goodbye"];;
```


Exemples (lists3.ml)

```
(* comparaison de listes *)
[1; 2] = [1; 2];;

[1; 2] < [1; 2; 3];;

[1; 3] < [1; 2; 3];;

[1; 2; 3] < [1; 3];; (* ordre lexicographique *)
```


Exemples (lists4.ml)

```
(* Constructeur de liste *)
1::[2;3];;

(* associe a droite *)
4::5::[6;7];;

(* erreur de typage *)
(4::5)::[6;7];;
```


Exemples (lists5.ml)

```
(* Destructeurs de listes *)

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

Fonctions de base sur les listes

- Les fonctions agissant sur les listes polymorphes ont des types polymorphes :
- L'opérateur `::` a le type : `' -> ' list -> ' list`
- La fonction `List.hd` a le type : `' list -> '`
- La fonction `List.tl` a le type : `' list -> ' list`
- Plus sur la polymorphie plus tard.
- `[]` et `::` sont les constructeurs de listes.

Autres fonctions utiles

- `List.length` donne la longueur d'une liste
- Opérateur `@` : concaténation de deux listes.
- Attention, `@` n'est pas un constructeur mais une fonction définie par récurrence.
- Plus de fonctions dans le module `List` de la bibliothèque standard.

Filtrage par motif

- angl. : pattern matching
- Très utile sur les type structurés, combinaison de
  - distinction de cas
  - un moyen facile de déconstruire une donnée
- Principe générale, pas seulement pour les listes.

Exemples (lists6.ml)

```
match [4; 5] with
| [] -> print_string "vide"
| a::l -> print_string "pas vide";;

let est_vide l = match l with
| [] -> true
| x -> false;;

let est_vide l = match l with
| [] -> true
| _ -> false;;
```

Exemples (lists7.ml)

```
let tete l = match l with
| [] -> failwith "Liste vide dans la fonction tete"
(* exception *)
| a::_ -> a;;

let queue l = match l with
| [] -> failwith "Liste vide dans la fonction queue"
(* exception *)
| _::finliste -> finliste;;

(* quel est le type de ces deux fonctions ? *)
```


Exemples (lists8.ml)

```
(* quel est l'erreur ? *)
let longueur l = match l with
| [] -> 0
| a::l' -> 1 + (longueur l');;
```


Les motifs

- Un motif est construit seulement d'identificateurs et de constructeurs.
- Si un motif s'applique, tous les identificateurs dans le motif sont liés. Leur portée : l'expression à droite du motif.
- On peut dans un motif écrire _ ou un identificateur préfixé par _, dans ce cas il n'y a pas de liaison.
- Les motifs doivent être linéaires (pas de répétition d'identificateur dans le même motif)

Filtrage par motif

- Les motifs sont essayés dans l'ordre donné.
- Le système OCaml vérifie qu'aucun cas n'a été oublié : l'ensemble des motifs doit être exhaustif.
- La non-exhaustivité donne lieu à un warning.
- Il est fortement conseillé de faire des distinctions de cas exhaustifs.

Exemples (lists9.ml)

```
let rec print_sep l = match l with
| [] -> print_newline()
| [toto] -> print_int toto
| titi::toto::finliste ->
    print_int titi;
    print_char ';';
    print_sep (toto::finliste);;

print_sep [1;2;3];;
```


Exemples (lists10.ml)

```
let rec even_length l =
  match l with
    | [] -> true
    | [_] -> false
    | _ :: _ :: reste -> even_length reste
;;

even_length [1;2;3];;
even_length [1;2;3;4];;
```


Exemples (lists11.ml)

```
(* erreur: motif non lineaire *)
let f l = match l with
  | x::x::reste -> 1
  | _ -> 0
;;
```


Exemples (lists12.ml)

```
(* erreur: pas un motif *)

let f l = match l with
  | 1+2::reste -> 3
  | _ -> 5
;;
```

Exemples (lists13.ml)

```
(* mauvais ordre de motifs *)

let f l = match l with
| [] -> print_string "vide"
| a::finliste -> print_int a
| [toto] -> print_int toto      (* jamais atteint *);;
;;
```


Exemples (lists14.ml)

```
 (* Filtrage non exhaustif *)
let g l = match l with
  | [] -> print_string "vide"
  | [toto] -> print_char toto;;

g ['a'];;

g ['a'; 'b'];;
```


Exemples (lists15.ml)

```
(* quel est l'erreur ? *)
let rec trouve a l = match l with
 | [] -> false
 | a::_ -> true
 | b::reste -> trouve a reste;;

trouve 1 [1;2;3];;
trouve 42 [1;2;3];;
```


Exemples (lists16.ml)

```
(* la fonction corrige *)
let rec trouve a l = match l with
 | [] -> false
 | b::r -> if a=b then true else trouve a r
;;

trouve 1 [1;2;3];;
trouve 42 [1;2;3];;
```

Exemples (motif2.ml)

```
(* motif avec des alternatives *)
let rec fib n = match n with
  | 0 | 1 -> n
  | n -> fib (n-1) + fib (n-2)
;;

fib 10;;
```


Exemples (motif3.ml)

```
(* la fonction trouve corrige avec when *)
let rec trouve a l = match l with
  | [] -> false
  | b::r when b=a -> true
  | _::r -> trouve a r
;;

trouve 1 [1;2;3];;
trouve 42 [1;2;3];;
```