Programmation Fonctionnelle
Cours 02

Michele Pagani

Université Paris Diderot
UFR Informatique
Laboratoire Preuves, Programmes et Systèmes

pagani@pps.univ-paris-diderot.fr

September 25, 2014

char

valeurs: caractères ASCII (écrits entre apostrophes ')
(*American Standard Code Information Interchange*)
'a', 'z', ' ','W'

échappement:

- \\ : antislash (\)
- \n : saut de ligne (line feed)
- \r : retour chariot (carriage return)
- \t : tabulation
- *ddd* : le caractère avec le code ASCII *ddd* en décimal
- \' : apostrophe (')

fonct. convers.:

- Char.code: char -> int
- Char.chr: int -> char
- Char.lowercase: char -> char
- Char.uppercase: char -> char

Voir manual (module `Char`) pour une liste complète.

char (exemples)

```
# 'a';;
- : char = 'a'

# Char.code 'a';;
- : int = 97

# '\097';;
- : char = 'a'

# '\97';;
    ^^
Error: Illegal backslash escape in string or character (\9)

# Char.uppercase 'a';;
- : char = 'A'

# Char.uppercase '[';;
- : char = '['
```

Un coup d'œil sur les modules

- Char.code appelle la fonction code du module Char.
- La bibliothèque standard de OCaml contient plusieurs modules qu'on utilisera dans la suite: Char, String, List, Array, ...
- pour appeler une fonction d'un module:
  - soit on écrit le nom du module suivi du nom de la fonction:

```
# Char.code;;
- : char -> int = <fun>
```

  - soit on ouvre le module avec le mot clé open et puis on appelle ses fonctions librement

```
# code;;
  ^^^
Error: Unbound value code
# open Char;;
# code;;
- : char -> int = <fun>
```

- on étudiera les modules en detail plus tard.

Un coup d'œil sur les modules

- Char.code appelle la fonction code du module Char.
- La bibliothèque standard de OCaml contient plusieurs modules qu'on utilisera dans la suite: Char, String, List, Array, ...
- pour appeler une fonction d'un module:
  - soit on écrit le nom du module suivi du nom de la fonction:

```
# Char.code;;
- : char -> int = <fun>
```

  - soit on ouvre le module avec le mot clé open et puis on appelle ses fonctions librement

```
# code;;
  ^^^
Error: Unbound value code
# open Char;;
# code;;
- : char -> int = <fun>
```

- on étudiera les modules en detail plus tard.

Un coup d'œil sur les modules

- Char.code appelle la fonction code du module Char.
- La bibliothèque standard de OCaml contient plusieurs modules qu'on utilisera dans la suite: Char, String, List, Array, ...
- pour appeler une fonction d'un module:
  - soit on écrit le nom du module suivi du nom de la fonction:

```
# Char.code;;
- : char -> int = <fun>
```

  - soit on ouvre le module avec le mot clé open et puis on appelle ses fonctions librement

```
# code;;
    ^^^
Error: Unbound value code
# open Char;;
# code;;
- : char -> int = <fun>
```

- on étudiera les modules en detail plus tard.

Un coup d'œil sur les modules

- Char.code appelle la fonction code du module Char.
- La bibliothèque standard de OCaml contient plusieurs modules qu'on utilisera dans la suite: Char, String, List, Array, ...
- pour appeler une fonction d'un module:
  - soit on écrit le nom du module suivi du nom de la fonction:

```
# Char.code;;
- : char -> int = <fun>
```

  - soit on ouvre le module avec le mot clé open et puis on appelle ses fonctions librement

```
# code;;
    ^^^
Error: Unbound value code
# open Char;;
# code;;
- : char -> int = <fun>
```

- on étudiera les modules en detail plus tard.

string

valeurs: chaînes de caractères (écrites entre guillemets ")
"Hello", "a", "␣","\097␣est␣a"

string ≠ char:
```
# "Hello".[1];;
- : char = 'e'

# "Hell"^'o';;
          ^^^
Error: This expression has type char but an
       expression was expected of type string
```

concatenation:
```
# "Hello␣"^"World";;
- : string = "Hello␣World"
```

autres fonct.:

- String.length: string -> int
- String.get: string -> int -> char
- String.make: int -> char -> string
- String.sub: string -> int -> int -> string

Voir manual (module `String`) pour une liste complète.

string (exemples)

```
# "\097␣est␣a";;
- : string = "a␣est␣a"

# "\097\"".[1];;
- : char = '"'

# let rec alphabet x =
        let x_str = String.make 1 x in
        let x_nxt = Char.chr (Char.code x +1) in
                if x = 'z' then x_str
                else x_str^(alphabet x_nxt);;
val alphabet : char -> string = <fun>

# alphabet 'a';;
- : string = "abcdefghijklmnopqrstuvwxyz"

# alphabet '\000';;
- : string = "\000\001\002\003\004\005\006\007\b\t\n\011\012
\r\014\015\016\017\018\019\020\021\022\023\024\025\026\027
\028\029\030\031␣!\"#$%&'()*+,-./0123456789:;<=>?@ABCDEFGHIJK
LMNOPQRSTUVWXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
```

Listes

type list

- des valeurs d'un même type peuvent être regroupées en listes:

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

# ['a';'b';'c'];;
- : char list = ['a'; 'b'; 'c']

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

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

- attention: tous les éléments de la même liste doivent être du même type

```
# [1; "deux"; 3];;
          ^^^
Error: This expression has type string but an expression
         was expected of type int
```

type list

- des valeurs d'un même type peuvent être regroupées en listes:

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

# ['a';'b';'c'];;
- : char list = ['a'; 'b'; 'c']

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

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

- attention: tous les éléments de la même liste doivent être du même type

```
# [1; "deux"; 3];;
        ^^^^
Error: This expression has type string but an expression
       was expected of type int
```

constructors de list

Une liste est soit vide soit a une tête et une queue

liste vide [] a un type polymorphe (⇒ plus tard)

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

- pour tout type a, il y a la liste vide [ ]
- 'a est une variable de type.

constructeur :: ajoute une tête a une queue

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

# 1::2::3::[];;                    (*associe a droite*)
- : int list = [1; 2; 3]

# 1::2::3;;
          ^^
Error: This expression has type int but an
       expression was expected of type int list
```

constructors de list

Une liste est soit vide soit a une tête et une queue

liste vide [] a un type polymorphe (⇒ plus tard)

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

- pour tout type a, il y a la liste vide [ ]
- 'a est une variable de type.

constructeur :: ajoute une tête a une queue

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

# 1::2::3::[];;                    (*associe a droite*)
- : int list = [1; 2; 3]

# 1::2::3;;
          ^^
Error: This expression has type int but an
       expression was expected of type int list
```

constructors de list

Une liste est soit vide soit a une tête et une queue

liste vide [] a un type polymorphe (⇒ plus tard)

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

- pour tout type a, il y a la liste vide [ ]
- 'a est une variable de type.

constructeur :: ajoute une tête a une queue

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

# 1::2::3::[];;                    (*associe a droite*)
- : int list = [1; 2; 3]

# 1::2::3;;
          ^^
Error: This expression has type int but an
       expression was expected of type int list
```

destructors de list

la "déstructuration" des listes (extraction des éléments) se fait par pattern-mathcing (⇒ plus tard)

```
# let f = function
    [] -> "vide"
    | t::q -> "pas␣vide";;
val f : 'a list -> string = <fun>

# let f x = match x with
   [] -> "vide"
   | t::q -> "pas␣vide";;
val f : 'a list -> string = <fun>

# f [ ];;
- : string = "vide"

# f [1;2];;
- : string = "pas␣vide"

# f ["toto"];;                    (*remarque polymorphisme*)
- : string = "pas␣vide"
```

list (exemples)

- on peut avoir plus cas que les deux constructors:

```
# let f = function
    [] -> "vide"
  | t::[] -> "singleton"
  | t::(tt::q) -> "au moins deux elements";;
val f : 'a list -> string = <fun>
```

- pour des exemples moins naïfs, il faut la récursion:

```
# let rec length l = match l with
      [] -> 0
    | t::q -> 1+length q;;
val length : 'a list -> int = <fun>

# length [1;2;3];;
- : int = 3

# length ["toto";"tata";"tutu"];;
- : int = 3
```

list (exemples)

- on peut avoir plus cas que les deux constructors:

```
# let f = function
    [] -> "vide"
    | t::[] -> "singleton"
    | t::(tt::q) -> "au moins deux elements";;
val f : 'a list -> string = <fun>
```

- pour des exemples moins naïfs, il faut la récursion:

```
# let rec length l = match l with
      [] -> 0
      | t::q -> 1+length q;;
val length : 'a list -> int = <fun>

# length [1;2;3];;
- : int = 3

# length ["toto";"tata";"tutu"];;
- : int = 3
```

list (quelques fonctions)

- @ : concaténation de deux listes (infix)

```
# [1]@[2;3];;
- : int list = [1; 2; 3]
# 1@[2;3];;
  ^^
Error: This expression has type int but an expression
       was expected of type 'a list
```

- d'autres fonctions dans le module List
  - List.hd : 'a list -> 'a
  - List.tl : 'a list -> 'a list
  - List.length : 'a list -> int

- les fonctions sur listes polymorphes ont types polymorphes

list (quelques fonctions)

- @ : concaténation de deux listes (infix)

```
# [1]@[2;3];;
- : int list = [1; 2; 3]
# 1@[2;3];;
  ^^
Error: This expression has type int but an expression
       was expected of type 'a list
```

- d'autres fonctions dans le module List
  - List.hd : 'a list -> 'a
  - List.tl : 'a list -> 'a list
  - List.length : 'a list -> int

- les fonctions sur listes polymorphes ont types polymorphes

list (quelques fonctions)

- @ : concaténation de deux listes (infix)

```
# [1]@[2;3];;
- : int list = [1; 2; 3]
# 1@[2;3];;
  ^^
Error: This expression has type int but an expression
       was expected of type 'a list
```

- d'autres fonctions dans le module List
  - List.hd : 'a list -> 'a
  - List.tl : 'a list -> 'a list
  - List.length : 'a list -> int

- les fonctions sur listes polymorphes ont types polymorphes

list (quelques fonctions)

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

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

# List.tl 1;;
          ^^
Error: This expression has type int but an expression
       was expected of type 'a list

# List.hd [];;                          (*exception, pas erreur*)
Exception: Failure "hd".

# List.tl [];;                          (*exception, pas erreur*)
Exception: Failure "tl".
```

list (quelques fonctions)

- Une implémentation possible de hd dans le module List :

```
let hd = function
  [] -> failwith "hd"
  | t::q -> t;;
val hd : 'a list -> 'a = <fun>

# hd [1;2;3];;
- : int = 1

# hd [];;
Exception: Failure "hd".
```

- le mécanisme d'exception permet de traiter les cas limites

(⇒ plus tard)

list (fonction map)

List.map : ('a -> 'b) -> 'a list -> 'b list

1. elle prend une fonction:
   f : 'a -> 'b
2. une liste :
   [e1; ... ;en] : 'a list
3. et renvoie la liste:
   [f(e1); ... ;f(en)] : 'b list

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]

# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

list (fonction map)

List.map : ('a -> 'b) -> 'a list -> 'b list

1. elle prend une fonction:
   f : 'a -> 'b
2. une liste :
   [e1; ... ;en] : 'a list
3. et renvoie la liste:
   [f(e1); ... ;f(en)] : 'b list

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]

# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

list (fonction map)

List.map : ('a -> 'b) -> 'a list -> 'b list

1. elle prend une fonction:
   f : 'a -> 'b
2. une liste :
   [e1; ... ;en] : 'a list
3. et renvoie la liste:
   [f(e1); ... ;f(en)] : 'b list

```
# List.map (function x->x+1) [3; 2; 6];;
- : int list = [4; 3; 7]

# List.map (function x->(x mod 2)=0) [1;4;6;3;8];;
- : bool list = [false; true; true; false; true]
```

list (itérateurs)

List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

1. elle prend une fonction:
   f : 'a -> 'b -> 'b
2. une liste de 'a:
   [e1; e2; ... ;en] : 'a list
3. un élément de 'b:
   x : 'b
4. et renvoie un élément de 'b:
   f e1 (f e2 ... (f en b)...) : 'b

```
# List.fold_right (fun x y -> x+y) [2;3;5] 1;;
- : int = 11

# List.fold_right (fun x y -> x+y) [] 1;;
- : int = 1
```

list (itérateurs)

List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

1. elle prend une fonction:
   f : 'a -> 'b -> 'b
2. une liste de 'a:
   [e1; e2; ... ;en] : 'a list
3. un élément de 'b:
   x : 'b
4. et renvoie un élément de 'b:
   f e1 (f e2 ... (f en b)...) : 'b

```
# List.fold_right (fun x y -> x+y) [2;3;5] 1;;
- : int = 11

# List.fold_right (fun x y -> x+y) [] 1;;
- : int = 1
```

list (itérateurs)

List.fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b

1. elle prend une fonction:
   f : 'a -> 'b -> 'b
2. une liste de 'a:
   [e1; e2; ... ;en] : 'a list
3. un élément de 'b:
   x : 'b
4. et renvoie un élément de 'b:
   f e1 (f e2 ... (f en b)...) : 'b

```
# List.fold_right (fun x y -> x+y) [2;3;5] 1;;
- : int = 11

# List.fold_right (fun x y -> x+y) [] 1;;
- : int = 1
```

list (itérateurs)

List . fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

1. elle prend une fonction:
   f : 'a -> 'b -> 'a
2. un élément de 'a:
   x : 'a
3. une liste de 'b:
   [e1; e2; ... ;en] : 'b list
4. et renvoie un élément de 'a:
   f (... (f (f x e1) e2) ...) en : 'a

```
# List.fold_right (fun x y -> x^y) ["Hello"; " "; "world"] "!";;
- : string = "Hello world!"

# List.fold_left (fun x y -> x^y) "!" ["Hello"; " "; "world"];;
- : string = "!Hello world"
```

list (itérateurs)

List . fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

1. elle prend une fonction:
   f : 'a -> 'b -> 'a
2. un élément de 'a:
   x : 'a
3. une liste de 'b:
   [e1; e2; ... ;en] : 'b list
4. et renvoie un élément de 'a:
   f (... (f (f x e1) e2) ...) en : 'a

```
# List.fold_right (fun x y -> x^y) ["Hello"; " "; "world"] "!";;
- : string = "Hello world!"

# List.fold_left (fun x y -> x^y) "!" ["Hello"; " "; "world"];;
- : string = "!Hello world"
```

list (itérateurs)

List . fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a

1. elle prend une fonction:
   f : 'a -> 'b -> 'a
2. un élément de 'a:
   x : 'a
3. une liste de 'b:
   [e1; e2; ... ;en] : 'b list
4. et renvoie un élément de 'a:
   f (... (f (f x e1) e2) ...) en : 'a

```
# List.fold_right (fun x y -> x^y) ["Hello"; "␣"; "world"] "!";;
- : string = "Hello␣world!"

# List.fold_left (fun x y -> x^y) "!" ["Hello"; "␣"; "world"];;
- : string = "!Hello␣world"
```

Pattern matching
(Filtrage par motifs)

Pattern matching

- fr.: filtrage par motif
- Très utile sur les types structurés, combinaison de
  - distinction de cas
  - un moyen facile de déconstruire une donnée

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Principe générale, pas seulement pour les listes: s'applique à n'importe quel type (sauf fonctions et objets)

```
# let rec fact n = match n with
  0 -> 1
  | n -> n* fact (n-1);;
val fact : int -> int = <fun>
```

Pattern matching

- fr.: filtrage par motif
- Très utile sur les types structurés, combinaison de
  - distinction de cas
  - un moyen facile de déconstruire une donnée

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Principe générale, pas seulement pour les listes: s'applique à n'importe quel type (sauf fonctions et objets)

```
# let rec fact n = match n with
  0 -> 1
  | n -> n* fact (n-1);;
val fact : int -> int = <fun>
```

Pattern matching

- fr.: filtrage par motif
- Très utile sur les types structurés, combinaison de
  - distinction de cas
  - un moyen facile de déconstruire une donnée

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Principe générale, pas seulement pour les listes: s'applique à n'importe quel type (sauf fonctions et objets)

```
# let rec fact n = match n with
  0 -> 1
  | n -> n* fact (n-1);;
val fact : int -> int = <fun>
```

Pattern matching

```
# let est_vide list = match list with
    [] -> true
    | t::q -> false;;
val est_vide : 'a list -> bool = <fun>
```

- pas nécessaire decomposition t::q :

```
# let est_vide list = match list with
    [] -> true
    | x-> false;;
val est_vide : 'a list -> bool = <fun>
```

- utilisation variable generique _ :

```
# let est_vide list = match list with
    [] -> true
    | _-> false;;
val est_vide : 'a list -> bool = <fun>
```

Pattern matching

```
# let est_vide list = match list with
    [] -> true
    | t::q -> false;;
val est_vide : 'a list -> bool = <fun>
```

- pas nécessaire decomposition t::q :

```
# let est_vide list = match list with
    [] -> true
    | x-> false;;
val est_vide : 'a list -> bool = <fun>
```

- utilisation variable generique _ :

```
# let est_vide list = match list with
    [] -> true
    | _-> false;;
val est_vide : 'a list -> bool = <fun>
```

Pattern matching

```
# let est_vide list = match list with
    [] -> true
    | t::q -> false;;
val est_vide : 'a list -> bool = <fun>
```

- pas nécessaire decomposition t::q :

```
# let est_vide list = match list with
    [] -> true
    | x-> false;;
val est_vide : 'a list -> bool = <fun>
```

- utilisation variable generique _ :

```
# let est_vide list = match list with
    [] -> true
    | _-> false;;
val est_vide : 'a list -> bool = <fun>
```

Pattern matching (exemples)

```
(* quel est le type de ces deux fonctions ? *)
let hd l = match l with
| [] -> failwith "Liste vide dans hd" (* exception *)
| a :: _ -> a ; ;

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


(* quel est l'erreur ?*)
let length l = match l with
| [] -> 0;
| t::q -> 1+ (length q);;
```

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Un pattern (en fr. 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 utiliser une variable générique _, dans ce cas il n'y a pas de liason
- Les motifs doivent être linéaires
  (pas de répétition d'identificateur dans le même motif)

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Un pattern (en fr. 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 utiliser une variable générique _, dans ce cas il n'y a pas de liason
- Les motifs doivent être linéaires
  (pas de répétition d'identificateur dans le même motif)

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Un pattern (en fr. 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 utiliser une variable générique _, dans ce cas il n'y a pas de liason
- Les motifs doivent être linéaires
  (pas de répétition d'identificateur dans le même motif)

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Un pattern (en fr. 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 utiliser une variable générique _, dans ce cas il n'y a pas de liason
- Les motifs doivent être linéaires
  (pas de répétition d'identificateur dans le même motif)

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Les motifs sont essayés dans l'ordre donné
- 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

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Les motifs sont essayés dans l'ordre donné
- 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

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Les motifs sont essayés dans l'ordre donné
- 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

Pattern matching (pattern)

```
# let rec map f list = match list with
    [] -> []
    | t::q -> (f t) :: (map f q);;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

- Les motifs sont essayés dans l'ordre donné
- 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

Pattern matching (exemples)

Les motifs doivent être linéaires:

```
# let rec even_length l = match l with
   [] -> true
   | [_] -> false
   | t::t:: reste -> even_length reste;;
          ^^
Error: Variable t is bound several times in this matching
```

```
# let rec even_length l = match l with
   [] -> true
   | [_] -> false
   | _::_:: reste -> even_length reste;;
val even_length : 'a list -> bool = <fun>

# even_length [1;2;3];;
- : bool = false

# even_length [1;2;3;4];;
- : bool = true
```

Pattern matching (exemples)

Les motifs doivent être linéaires:

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | t::t:: reste -> even_length reste;;
        ^^
Error: Variable t is bound several times in this matching
```

```
# let rec even_length l = match l with
    [] -> true
  | [_] -> false
  | _::_:: reste -> even_length reste;;
val even_length : 'a list -> bool = <fun>

# even_length [1;2;3];;
- : bool = false

# even_length [1;2;3;4];;
- : bool = true
```

Pattern matching (exemples)

Un motif est construit seulement d'identificateurs et de constructeurs:

```
# let rec fact n = match n with
  0 -> 1
  |n+1 -> (n+1)*(fact n);;
   ^^^
Error: Syntax error

# let rec length list = match list with
  [] -> 0
  |[t]@q -> 1+length q;;
     ^^^
Error: Syntax error
```

Pattern matching (exemples)

Les motifs sont essayés dans l'ordre donné

```
# let rec fact n = match n with
   n -> n* fact (n-1)
   | 0 -> 1;;
Warning 11: this match case is unused.
val fact : int -> int = <fun>

# fact 2;;
Stack overflow during evaluation (looping recursion?).
```

Pattern matching (exemples)

L'ensemble des motifs doit être exhaustif

```
# let hd list = match list with
  t::q -> t;;

Warning 8: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val hd : 'a list -> 'a = <fun>

# hd [];;
Exception: Match_failure ("//toplevel//", 113, -6).
```

Pattern matching (exemples)

```
(* quel est l'erreur ? *)

# let rec trouve a list = match list with
    [] -> false
  |a::_-> true
  |b::q -> trouve a q;;
val trouve : 'a -> 'b list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = true
```

Tous les identificateurs dans le motif sont liés

Pattern matching (exemples)

```
(* quel est l'erreur ? *)

# let rec trouve a list = match list with
    [] -> false
  |a::_-> true
  |b::q -> trouve a q;;
val trouve : 'a -> 'b list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = true
```

Tous les identificateurs dans le motif sont liés

Pattern matching (exemples)

```
(* la fonction trouve corrige *)

# let rec trouve a list = match list with
    [] -> false
  |b::q -> if b=a then true else trouve a q;;
val trouve : 'a -> 'a list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true

# trouve 42 [1;2;3];;
- : bool = false
```

Pattern matching (exemples)

- patterns avec des alternatives

```
let rec fib n = match n with
0 | 1 -> n
  | n -> fib (n-1)+fib (n-2);;
val fib : int -> int = <fun>

# fib 10;;
- : int = 55
```

- patterns avec des conditions

```
let rec trouve a list = match list with
  [] -> false
  | b::q when b=a -> true
  | _::q -> trouve a q;;
val trouve : 'a -> 'a list -> bool = <fun>

# trouve 1 [1;2;3];;
- : bool = true
# trouve 42 [1;2;3];;
- : bool = false
```

Doggy bag

- types char et string
  - concatenation string ^
  - quelques fonctions dans modules Char, String
    code, cur, lowercase, ..., length, get, make, sub, ...
- type list :
  - type polymorphe 'a list
  - constructeurs [], :: / "déstructuration" par filtrage par motifs
  - quelques fonctions:
    - concaténation @
    - dans module List :
      hd, tl, length, map, fold_right, fold_left
- filtrage par motifs (pattern matching)
  - distinction de cas / déstructuration donnée
  - motifs
    (identificateurs/constructeurs, liaison, linéarité, ordre d'évaluation, exhaustivité)
  - motifs avec alternatives, conditions

Doggy bag

- types char et string
  - concatenation string ^
  - quelques fonctions dans modules Char, String
    code, cur, lowercase, ..., length, get, make, sub, ...
- type list :
  - type polymorphe 'a list
  - constructeurs [], :: / "déstructuration" par filtrage par motifs
  - quelques fonctions:
    - concaténation @
    - dans module List :
      hd, tl, length, map, fold_right, fold_left
- filtrage par motifs (pattern matching)
  - distinction de cas / déstructuration donnée
  - motifs
    (identificateurs/constructeurs, liaison, linéarité, ordre d'évaluation, exhaustivité)
  - motifs avec alternatives, conditions

Doggy bag

- types char et string
  - concatenation string ^
  - quelques fonctions dans modules Char, String
    code, cur, lowercase, ..., length, get, make, sub, ...
- type list :
  - type polymorphe 'a list
  - constructeurs [], :: / "déstructuration" par filtrage par motifs
  - quelques fonctions:
    - concaténation @
    - dans module List :
      hd, tl, length, map, fold_right, fold_left
- filtrage par motifs (pattern matching)
  - distinction de cas / déstructuration donnée
  - motifs
    (identificateurs/constructeurs, liaison, linéarité, ordre d'évaluation, exhaustivité)
  - motifs avec alternatives, conditions

Eh bien, maintenant vous pouvez le comprendre !

```
1  let rec qSort list =
2   match list with
3   | [] -> []
4   | pivot::rest ->
5    split pivot [] [] rest
6  and split pivot left right list =
7   match list with
8   | [] -> (qSort left)@( pivot :: (qSort right))
9   | hd :: tl ->
10   if hd <= pivot then split pivot (hd :: left) right tl
11   else split pivot left (hd :: right) tl;;
```