[PF5] Cours 1 22/9/14

“

michele.pagani@pps.univ-paris-diderot.fr

Char

1. Valeurs

Caractères ASCII `'a', 'z', ' ', 'W'`

2. Échapement

- `\` : antislash ()
- `\n` : Saut de ligne
- `\r` : retour chariot
- `\t` : tabulation
- `\ddd` : char avec le code ASCII `ddd` en décimal

- `\'` : apostrophe

3. Fonction de conversion

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

Voir le module `Char` pour une liste plus complète.

3.1. 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 = '['
```

Module (Intro)

- `Char.code` appelle la fonction code du module **Char**
- La bibliothèque standard de `OCaml` contient pusieurs modules qu'on utilisera par la suite: `Char, String, List, Array, ...`
- Pour appeler une fonction d'un module :
    - Soit on ecrit le nom du module suivi du nom de la fonction:

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

- Soit ont ouvre le module avec `open nom_Module` puis on appelle les fonctions librement

```
# code;;
  ^^^
Error: Unbound value Code
```

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

String

1. valeurs

Chaîne de caractères(entre guillemets ") `"Hello", "a", " ", "\097 est a"`

2. String ≠ char

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

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

3. Concatenation

```
# "Hello "^"World";;
- : String = "Hello World"
```

4. Fonctions

- `String.length` : `string -> int` Donne la taille de la chaîne
- `String.get` : `string -> int -> char` donne le char la à la i-ème position
- `String.make` : `int -> char -> string` donne un string conenant n char c

- `String.sub` : `string -> int -> int -> string` **Voir le module** `Char` **pour une liste plus complète.**

5. Exemple

```
# "\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:;<=>?@ABCDEFGHIJKLMNOPQRSTUV
WXYZ[\\]^_`abcdefghijklmnopqrstuvwxyz"
```

Liste

- Des valeurs de **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) liste = [<fun>; <fun>]

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

- **Attention:** tout 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
```

1. Constructeur

Une liste est soit **vide**, soit a une **tête** et une **queue** liste vide `[]` a un type **polymorphe** *voir plus tard*

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

- Pour tout type `a`, il y a une 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
```

2. Destructuration

La "destructuration" des listes (extraction des éléments) se fait par **patter-matching**

```
# 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"
```

3. Exemples

- On peut avoir plus de cas que les deux constructeur:

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

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

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

4. Quelques fonctions

- `@` : Concaténation de deux **liste** (*infixe*)

```
# [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 **Liste**
  - `List.hd` : `'a list -> 'a` (*retourne la tête de la liste*)
  - `List.tl` : `'a list -> 'a list` (*retourne la queue de la liste*)
  - `List.length` : `'a list -> int` (*retourne la taille de la liste*)
- Les fonction sur les listes polymorphe sont de type polymorphe

```
# 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 int list

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

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

- 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 (*voir plus tard*)

4.1. 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]
```

4.2. List Itérateurs

List.fold_right

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

1. Elle prend une fonction `f: 'a -> 'b -> 'b`
2. Une liste : `[e1; ...; 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;5;6] 1;;
- : int = 11

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

List.fold_left

```
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. 1. Une liste : `[e1; ...; en] : 'b list`
4. Et renvoie un élément de `'a` : `f (... (f (f x e1) e2) ...) : '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

- fr.: **filtrage par motif**
- Très utile sur les types structurés, combinaison de
  - distinction des 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 aussi à n'importe quel type **(sauf fonction et objet)**

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