

Algorithmique et Approche Fonctionnelle

Cours 4 : Les Listes

06 Octobre 2008

Le type des séquences (listes)

Séquences d'entiers

On peut utiliser un type somme pour représenter des séquences (non bornées) d'entiers

```
# type int_list = Nil | Cons of int * int_list;;
```

- Nil représente la séquence vide
- Cons(x,l) est la séquence dont le premier élément (on dit aussi la **tête**) est x et la **suite** est la séquence l

par exemple, la séquence 4 ; 1 ; 5 ; 8 ; 1 est représentée par la valeur

```
# Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil)))));;  
- : int_list = Cons(4,Cons(1,Cons(5,Cons(8,Cons(1,Nil))))))
```

le type int list

le type des séquences est prédéfini en OCAML et ses éléments se notent avec une syntaxe spéciale

- Cons se note :: et est infixe
- Nil se note []

par exemple, la séquence 4 ; 1 ; 5 ; 8 ; 1 est représentée par la valeur

```
# 4::1::5::8::1:: [];;  
- : int list = [4;1;5;8;1]
```

on peut aussi directement utiliser la notation [e1 ; e2 ; ... ; en]

```
# [4;1;5;8;1];;  
- : int list = [4;1;5;8;1]
```

ou faire un mélange des deux notations

```
# 4::1:: [5;8;1];;  
- : int list = [4;1;5;8;1]
```

OCAML permet de définir des listes dont les éléments peuvent être autre chose que des entiers

```
# ['c' ; 'a' ; 'm' ; 'l'];;
- : char list = ['c' ; 'a' ; 'm' ; 'l']
```

```
# [ ["des" ; "listes"] ; ["de" ; "listes"] ];;
- : string list list = [ ["des" ; "listes"] ; ["de" ; "listes"] ]
```

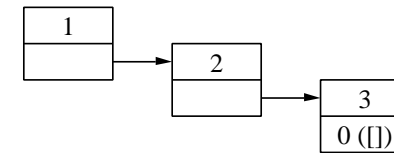
mais il n'est pas possible de construire une liste d'éléments de types différents

```
# [10 ; 'a' ; 4];;
---
This expression has type char but is here used with type int
```

les listes prédéfinies en OCAML correspondent exactement aux **listes chaînées** définies habituellement en C par le type suivant

```
typedef struct list{
    int elt;
    struct list* suivant;
};
```

la représentation mémoire de ces listes correspond à un chaînage de blocs mémoire, par exemple, la liste **[1 ; 2 ; 3]** correspond à



accès aux éléments d'une liste

on accède aux éléments d'une liste à l'aide des fonctions prédéfinies **List.hd** et **List.tl**

```
# List.hd [3;6;1;2];;
- : int = 3
# List.tl [3;6;1;2];;
- : int list = [6;1;2];;
```

List.hd et List.tl échouent sur une liste vide

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

# List.tail [];;
Exception : Failure "tl".
```

Fonctions sur les listes

la définition des fonctions sur les listes prennent généralement la forme d'une définition à deux cas

- le cas où la liste est vide
- le cas où elle ne l'est pas

Pour cette raison, il est plus agréable de réaliser cette analyse par cas avec du filtrage

```
# let f l =
  match l with
  [] -> ...
  | x::s -> ...
```

la fonction zeros vérifie que tous les éléments d'une liste d'entiers sont des 0 (renvoie true si la liste est vide)

```
# let rec zeros l =
  match l with
  [] -> true
  | x::s -> x=0 && zeros s ;;
val zeros : int list -> bool = <fun>
```

```
# zeros [];;
- : bool = true
```

```
# zeros [0;0;0];;
- : bool = true
```

```
# zeros [0;1;0];;
- : bool = false
```

évaluation de la fonction zeros

évaluation de zeros [0;0;0]

[0;0;0] ≠ [], x = 0, s = [0;0]	⇒	0=0 && zeros [0;0]
	=	zeros [0;0]
[0;0] ≠ [], x = 0, s = [0]	⇒	0=0 && zeros [0]
	=	zeros [0]
[0] ≠ [], x = 0, s = []	⇒	0=0 && zeros []
	=	zeros []
[] = []	⇒	true

évaluation de zeros [0;1;0]

[0;1;0] ≠ [], x = 0, s = [1;0]	⇒	0=0 && zeros [1;0]
	=	zeros [1;0]
[1;0] ≠ [], x = 1, s = [0]	⇒	1=0 && zeros [0]
	=	false

recherche d'un entier dans une liste

la fonction recherche détermine si un entier n figure bien dans une liste l

```
# let rec recherche n l =
  match l with
  [] -> false
  | x::s -> x=n || recherche n s
val recherche : int list -> bool = <fun>
```

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

```
# recherche 4 [1;2];;
- : bool = false
```

évaluation de recherche 4 [3;2;4;1]

```

recherche 4 [3;2;4;1]
[3;2;4;1] ≠ [], x = 3, s = [2;4;1] ⇒ 3=4 || recherche 4 [2;4;1]
= recherche 4 [2;4;1]
[2;4;1] ≠ [], x = 2, s = [4;1] ⇒ 2=4 || recherche 4 [4;1]
= recherche 4 [4;1]
[4;1] ≠ [], x = 4, s = [1] ⇒ 4=4 || recherche 4 [1]
= true

```

évaluation de recherche 4 [1;2]

```

recherche 4 [1;2]
[1;2] ≠ [], x = 1, s = [2] ⇒ 1=4 || recherche 4 [2]
= recherche 4 [2]
[2] ≠ [], x = 2, s = [] ⇒ 2=4 || recherche 4 []
= recherche 4 []
[] = [] ⇒ false

```

la fonction longueur retourne la longueur d'une liste

```

# let rec longueur l =
  match l with
  [] -> 0
  | _::s -> 1 + (longueur s);;

```

une version récursive terminale

```

# let longueur l =
  let rec longrec acc l =
    match l with
    [] -> acc
    | _::s -> longrec (1+acc) s
  in
  longrec 0 l

```

cette fonction est prédéfinie en OCAML : List.length

évaluation de longueur [10;2;4]

```

longueur [10;2;4]
= longrec 0 [10;2;4]
[10;2;4] ≠ [], s = [2;4] ⇒ longrec (1+0) [2;4]
[2;4] ≠ [], s = [4] ⇒ longrec (1+1) [4]
[4] ≠ [], s = [] ⇒ longrec (1+2) []
[] = [] ⇒ 3

```

quel est le type de la fonction longueur ?

longueur doit pouvoir s'appliquer à des listes d'entiers, comme par exemple

```

# longueur [4;3;6;1;10];;
- : int = 5

```

... alors elle doit avoir le type suivant

```

val longueur : int list -> int

```

mais cette fonction doit aussi pouvoir être appliquée sur une liste dont les éléments sont d'un autre type, comme par exemple

```

# longueur [[4.5;0.3;9.8];[];[3.2;1.8]];;
- : int = 3

```

... dans ce cas la fonction longueur devrait également avoir comme type

```

val longueur : (float list) list -> int

```

- les deux types précédents sont corrects
- la fonction longueur a une **infinité** de types

le type inféré par OCAML est le plus général

```
val longueur : 'a list -> int
```

'a (qui se lit "apostrophe a", ou encore "alpha") est une **variable de type**

une variable de type veut dire **n'importe quel type**

il faut donc lire le type de la fonction longueur comme suit

la fonction longueur prend en argument une **liste** – dont les éléments sont de **n'importe quel type** – et retourne un **entier**

Fonctions génériques sur les listes

concaténation de listes

la fonction append construit une nouvelle la liste en réunissant deux listes bout à bout

```
# let rec append l1 l2 =
  match l1 with
  | [] -> l2
  | x::s -> x::(append s l2);;
```

```
val append : 'a list -> 'a list -> 'a list = <fun>
```

```
# append [2;5;1] [10;6;8;15];;
```

```
- : int list = [2;5;1;10;6;8;15]
```

- cette fonction est prédéfinie en OCAML, il s'agit de `List.append`
- l'opérateur infixe `@` est un raccourci syntaxique pour cette fonction, on note `l1@l2` la concaténation de l1 et l2

évaluation de append

évaluation de append [1;2] [3;4]

	append [1;2] [3;4]
[1;2] ≠ [], x = 1, s = [2]	⇒ 1::(append [2] [3;4])
[2] ≠ [], x = 2, s = []	⇒ 1::2::(append [] [3;4])
[] = []	⇒ 1::2::[3;4]
	⇒ 1::[2;3;4]
	⇒ [1;2;3;4]

- la fonction `append` n'est pas récursive terminale
- si l'ordre des éléments n'a pas d'importance, on peut définir une concaténation récursive terminale qui inverse les éléments de la première liste

```
let rec rev_append l1 l2 =
  match l1 with
  [] -> l2
  | x :: s -> rev_append s (x :: l2)
val rev_append : 'a list -> 'a list -> 'a list = <fun>
```

```
# rev_append [4;2;6] [1;10;9;5];;
- : int list = [6; 2; 4; 1; 10; 9; 5]
```

la fonction `rev` pour renverser une liste `l` s'obtient facilement en concaténant la liste `l` avec la liste vide `[]`, en utilisant `rev_append`

```
# let rev l = rev_append l [];;
val rev : 'a list -> 'a list = <fun>
```

```
# rev [4;2;6;1];;
- : int list = [1; 6; 2; 4]
```

évaluation de `rev [1;2;3]`

	<code>rev [1;2;3]</code>
	<code>= rev_append [1;2;3] []</code>
<code>[1;2;3] ≠ [], x = 1, s = [2;3]</code>	<code>⇒ rev_append [2;3] (1::[])</code>
	<code>= rev_append [2;3] [1]</code>
<code>[2;3] ≠ [], x = 2, s = [3]</code>	<code>⇒ rev_append [3] (2::[1])</code>
	<code>= rev_append [3] [2;1]</code>
<code>[3] ≠ [], x = 3, s = []</code>	<code>⇒ rev_append [] (3::[2;1])</code>
	<code>= rev_append [] [3;2;1]</code>
<code>[] = []</code>	<code>⇒ [3;2;1]</code>