
Quelques fonctions récursives de base sur les listes et les arbres

Fonctions sur les listes

La longueur d'une liste

```
# let rec longueur l = match l with
    [] -> 0
    | a :: r -> 1 + longueur r ;;
val longueur : 'a list -> int = <fun>
```

```
# longueur [] ;;
- : int = 0
```

```
# longueur [1;4;2;5;3;7] ;;
- : int = 6
```

Exécution de la fonction longueur

longueur [1;4;2;5;3;7]

1 + longueur [4;2;5;3;7]

1 + longueur [2;5;3;7]

1 + longueur [5;3;7]

1 + longueur [3;7]

1 + longueur [7]

1 + longueur []

0

--> 6

La concatenation

```
# let rec concat l1 l2 = match l1 with
    [] -> l2
  | a::r -> a:: concat r l2;;
val concat : 'a list -> 'a list -> 'a list = <fun>

# concat [] [] ;;
- : '_a list = []
# concat [] [2;3;4];;
- : int list = [2; 3; 4]

# concat [7;8;9] [4;5;6];;
- : int list = [7; 8; 9; 4; 5; 6]
```

Exécution de la fonction concat

concat [7;8;9] [4;5;6]

7 :: concat [8;9] [4;5;6]

8 :: concat [9] [4;5;6]

9 :: concat [] [4;5;6]

[4;5;6]

-->

[7;8;9;4;5;6]

L'appartenance

```
# let rec appartient e l = match l with
  []    -> false
  | p::r -> p=e or appartient e r;;
val appartient : 'a -> 'a list -> bool = <fun>
```

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

La sous-liste qui commence à la position n

Cette fonction est définie pour toute position n d'une liste l telle que $1 \leq n \leq \text{longueur } l$.

```
# let rec sous_liste l n = match l with
  []      -> failwith "erreur"
  | _::r   -> if n=1 then l else sous_liste r (n-1);;
```

```
# sous_liste [3;4] 0 ;;
```

```
Exception: Failure "erreur".
```

```
# sous_liste [3;4] 1 ;;
```

```
- : int list = [3;4]
```



```
# sous_liste [3;4] 2 ;;  
- : int list = [4]  
# sous_liste [3;4] 3 ;;  
Exception: Failure "erreur".
```

Exécution de la fonction sous-liste

```
sous_liste [22;32;41] 3
```

```
sous_liste [32;41] 2
```

```
sous_liste [41] 1
```

```
-->
```

```
[41]
```

Le parcours d'une liste (I)

Rajouter 5 à chaque élément d'une liste d'entiers

```
# let rec raj5 l = match l with
    [] -> []
    | p::r -> p+5 :: raj5 r;;
val raj5 : int list -> int list = <fun>

# raj5 [1;2;3];;
- : int list = [6; 7; 8]
```

Exécution de la fonction raj5

raj5 [1;2;3]

1+5 :: raj5 [2;3]

2+5 :: raj5 [3]

3+5 :: raj5 []

[]

->

[6;7;8]

Le parcours d'une liste (II)

Concatener le mot "abc" devant chaque mot d'une liste de mots.

```
# let rec concatabc l = match l with
  []      -> []
  | p::r  -> ("abc"^p)::concatabc r;;
val concatabc : string list -> string list = <fun>

# concatabc ["df"; "gh"];;
- : string list = ["abcdf"; "abcgh"]
```

Le parcours d'une liste avec la fonction d'ordre supérieur map

```
# let rec map f l = match l with
  []      -> []
  | p::r  -> (f p) :: map f r ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

```
# raj5 [1;2;3];;
- : int list = [6; 7; 8]
```

```
# map (fun x -> x+5) [1;2;3] ;;
- : int list = [6; 7; 8]
```

```
# concatabc ["df"; "gh"];;  
- : string list = ["abcdf"; "abcgh"]  
  
# map (fun x -> "abc"^x) ["df"; "gh"];;  
- : string list = ["abcdf"; "abcgh"]  
  
# map (function (x,y) -> x) [(1,2);(3,2);(4,2)];;  
- : int list = [1; 3; 4]  
  
# map (function (x,y) -> y ) [(1,2);(3,2);(4,2)];;  
- : int list = [2; 2; 2]
```

Fonctions sur les arbres

Les arbres binaires d'entiers

```
# type abe = Av | N of int * abe * abe;;
```

```
type abe = Av | N of int * abe * abe
```

```
# let a0 = Av;;
```

```
# let a1 = N(1,Av,Av);;
```

```
# let a2 = N(1,N(2,Av,Av),Av);;
```

```
# let a3 = N(1,Av,N(3,Av,Av));;
```

```
# let a4 = N(1,N(2,Av,Av),N(3,Av,Av));;
```


Profondeur d'un arbre

```
# let rec prof a = match a with
    Av          -> 0
    | N(_,x,y) -> 1 + max (prof x) (prof y);;
val prof : abe -> int = <fun>
```

```
# prof a0;;
- : int = 0
# prof a1;;
- : int = 1
# prof a2;;
- : int = 2
```

```
# prof a3;;  
- : int = 2  
# prof a4;;  
- : int = 2
```

Exécution de la fonction prof

```
      prof N(1,N(2,Av,Av),N(3,Av,Av))
            /                \
    prof N(2,Av,Av)          prof N(3,Av,Av)
        /      \              /      \
    prof Av    Prof Av      Prof Av    Prof Av
```

Nombre de feuilles d'un arbre

```
# let rec nb_feuilles a = match a with
  Av          -> 1
  | N(_,x,y) -> nb_feuilles x + nb_feuilles y;;
val nb_feuilles : abe -> int = <fun>
```

```
# nb_feuilles a0;;
```

```
- : int = 1
```

```
# nb_feuilles a1;;
```

```
- : int = 2
```

```
# nb_feuilles a2;;
```

```
- : int = 3
```

```
# nb_feuilles a3;;
```

```
- : int = 3
```

```
# nb_feuilles a4;;
```

```
- : int = 4
```

Nombre de noeuds internes d'un arbre

```
# let rec nb_noeuds_internes a = match a with
  Av          -> 0
  | N(_,x,y)  -> 1+  nb_noeuds_internes x +
                    nb_noeuds_internes y;;
val nb_noeuds_internes : abe -> int = <fun>

# nb_noeuds_internes a0;;
- : int = 0
# nb_noeuds_internes a1;;
- : int = 1
# nb_noeuds_internes a2;;
- : int = 2
```



```
# nb_noeuds_internes a3;;  
- : int = 2  
# nb_noeuds_internes a4;;  
- : int = 3
```

Nombre de noeuds d'un arbre

```
# let rec nb_noeuds a = match a with
  Av          -> 1
  | N(_,x,y) -> 1+ nb_noeuds x + nb_noeuds y;;
val nb_noeuds : abe -> int = <fun>
```

```
# nb_noeuds a0;;
```

```
- : int = 1
```

```
# nb_noeuds a1;;
```

```
- : int = 3
```

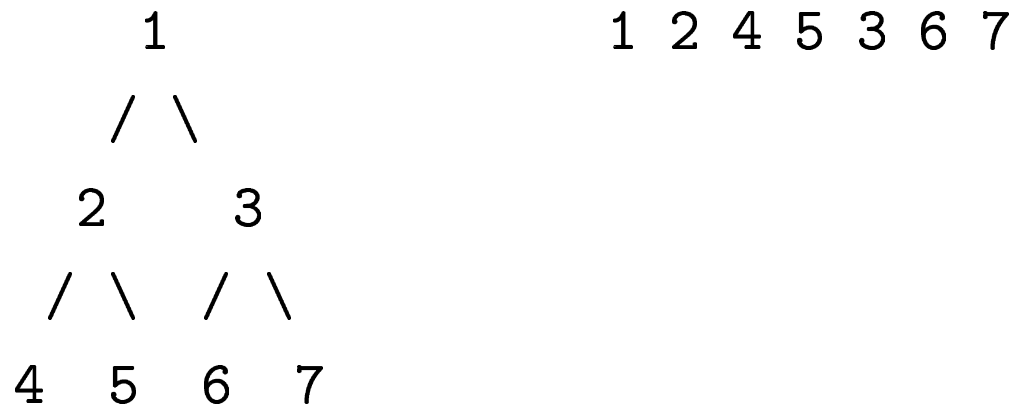
```
# nb_noeuds a2;;
```

```
- : int = 5
```

```
# nb_noeuds a3;;  
- : int = 5  
# nb_noeuds a4;;  
- : int = 7
```

Parcours préfixe d'un arbre

On veut le parcours suivant :



Sens du parcours : **Noeud - Fils gauche - Fils droit**

Parcours préfixe en OCAML

```
# let rec parcours_prefixe a = match a with
  Av          -> ()
  | N(e,x,y) -> print_int e ;
                (parcours_prefixe x) ;
                (parcours_prefixe y);;
```

```
# let b1 = N(1,N(2,
                N(4,Av,Av),
                N(5,Av,Av)),
            N(3,
                N(6,Av,Av),
                N(7,Av,Av)));;
```

```
# parcours_prefixe b1;;  
1245367- : unit = ()
```

Parcours infixe d'un arbre

On veut le parcours suivant :



Sens du parcours : **Fils gauche - Noeud - Fils droit**

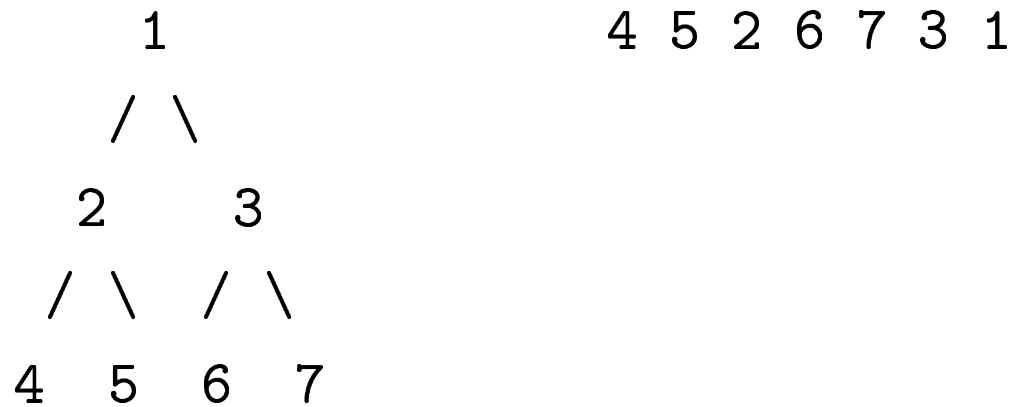
Parcours infixe en OCAML

```
# let rec parcours_infixe a = match a with
  Av          -> ()
  | N(e,x,y) -> (parcours_infixe x) ;
                print_int e ;
                (parcours_infixe y);;

# parcours_infixe b1;;
4251637- : unit = ()
```


Parcours suffixe d'un arbre

On veut le parcours suivant :



Sens du parcours : Fils gauche - Fils droit - Noeud

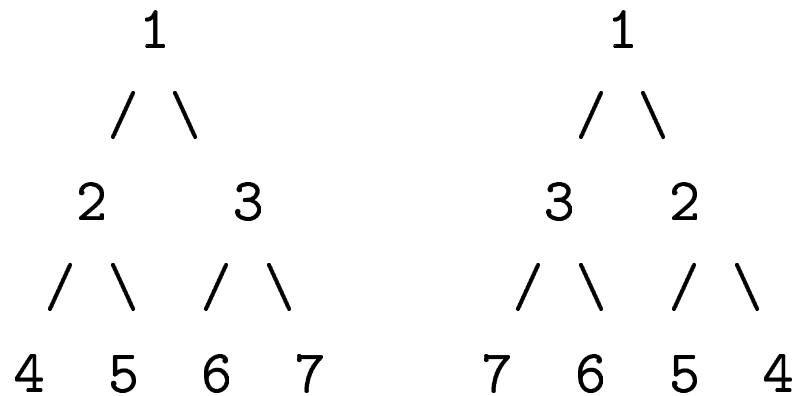
Parcours suffixe en OCAML

```
# let rec parcours_suffixe a = match a with
  Av          -> ()
  | N(e,x,y) -> (parcours_suffixe x);
                (parcours_suffixe y);
                (print_int e) ;;
```

```
# parcours_suffixe b1;;
4526731- : unit = ()
```

Miroir d'un arbre

On veut la transformation suivante :



```
# let rec miroir a = match a with
    Av -> Av
    | N(e,x,y)-> N(e, miroir y , miroir x);;
val miroir : abe -> abe = <fun>
```

```
# miroir a4;;  
- : abe = N (1, N (3, Av, Av), N (2, Av, Av))
```

```
# miroir b1;;  
- : abe = N (1,  
            N (3,  
              N (7, Av, Av),  
              N (6, Av, Av)),  
            N (2,  
              N (5, Av, Av),  
              N (4, Av, Av)))
```