Informations

partiel 27 mars de 12h30 a 15h30 en amphi 43

projet le sujet devrait être distribué la semaine du 22 mars

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
```

Remarques

- On "voit" bien que la fonction termine :
  - la longueur de la liste dans l'appel recursif diminue de 1 à chaque appel

  - Une fois l'appel recursif terminé, il reste encore du travail a faire pour obtenir le resultat (le `1+ ...`)
    Cela oblige l'interprète à garder en suspens une longue série de calculs, ce qui occupe de la mémoire précieuse

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
```

Éviter les calculs en suspens : la récursion terminale

On peut aisément écrire la fonction `longueur` de sorte à éviter de garder des longs calculs en suspens : pour cela, on utilise des *accumulateurs* qui gardent le résultat partiel d'un calcul, et on écrit les fonctions en s'assurant que chaque appel récursif soit terminal[1].
Ce style de programmation porte à introduire des fonctions auxiliaries.

```
let rec acc_long n =
     function
       []   -> n
     | a::r -> acc_long (n+1)
;;

let longueur l = acc_long 0
;;
```

Exécution de la fonction `acc_long`

```
(acc_long 0 [1;4;2;5;3;7])
(acc_long 1 [4;2;5;3;7])
(acc_long 2 [2;5;3;7])
(acc_long 3 [5;3;7])
```

[1] C.à.d. : une fois l'appel récursif fait,on ne fait plus d'autres calculs, et on renvoye directement le résultat reçu.

```
(acc_long 4 [3;7])
(acc_long 5 [7])
(acc_long 6 [])
--> 6
```

Avantages de la recursion terminale

Comme on ne garde pas de calculs en suspens, on peut transformer le programme récursif en une boucle (plus efficace) ; le compilateur Ocaml le fait. Un certain nombre de fonctions de la libraire standard Ocaml sont écrite de cette façon, par exemple, `List.length` et `List.rev`.

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 (fun (x,y) -> x) [(1,2);(3,2);(4,2)];;
- : int list = [1; 3; 4]

# map (fun (x,y) -> y ) [(1,2);(3,2);(4,2)];;
- : int list = [2; 2; 2]
```

Quelques fonctions de la librarie

```
# open List;;
# length;;
- : 'a list -> int = <fun>
# append;;
- : 'a list -> 'a list -> 'a list = <fun>
# [1;2]@[3;4];;
- : int list = [1; 2; 3; 4]
# filter;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# filter (fun x -> x >10) [1;23;15;2;6;9];;
- : int list = [23; 15]
```

```
# sort;;
- : (’a -> ’a -> int) -> ’a list -> ’a list = <fun>
# sort (fun x -> fun y -> x-y) [1;4;2;3;5;7;6];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
# sort compare [1;4;2;3;5;7;6];;
- : int list = [1; 2; 3; 4; 5; 6; 7]
```

Défi

Comment trouver tous les anagrammes dans un vrai dictionnaire Français en temps raisonnable ? Ma solution fait 52 lignes et met 9 secondes[2] pour traiter 138258 mots sur un Pentium III à 800Mhz. Et la votre ?

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
```

[2] 4 avec la version native


```
    | 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

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 :

```
    1             1 2 4 5 3 6 7
   / \
  2   3
 / \ / \
4  5 6  7
```

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 :

```
    1             4 2 5 1 6 3 7
   / \
  2   3
 / \ / \
4  5 6  7
```

Sens du parcours : Fils gauche - Noeud - Fils droit

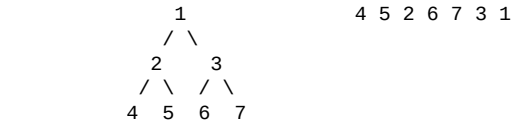
Parcours infixe en OCAML

```
# 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 :



4 5 2 6 7 3 1

Sens du parcours : Fils gauche - Fils droit - Noeud

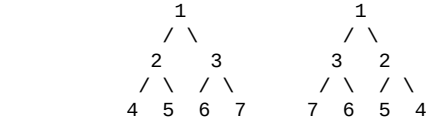
Parcours suffixe en OCAML

```
# 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)))
```