
```
(* Exo 2 : produit*)
let rec produit = fun
   |[] -> 1
   |(a::q) ->  a*(produit q);;
Une version terminale :
let rec produit_term l p= match l with
   |[] -> p
   |(a::q) -> produit_term q (a*p);;
```

```
(*Exo 3 : double*)
let rec double = fun
 |[] -> []
  |(a::q) -> a::a::(double q);;
```

```
(*Exercice 4 : avant-dernier*)
let rec avantdernier = fun
   |[] -> failwith "liste de moins de deux éléments"
   | [a] -> failwith "liste de moins de deux éléments"
  |(a::[b]) -> a
  |(a::q) -> avantdernier q;;

signature : 'a list -> 'a
```

(*Exo 5 : mystere*)
la fonction mystere renvoie le premier entier naturel qui ne figure pas dans la liste en argument.

```
(* Exo 6 : somme*)

let rec somme = fun
   |[] _ -> []
  |_ [] -> []
   |l m ->(hd l +hd m)::(somme (tl l) (tl m));;
```

```
(*Exo 7 : polynomes*)
1-
let p=[(0,9);(3,8);(102,5)];;

2-
let rec add  p1 p2 = match (p1,p2) with
   |[] ,_-> p2
   |_,[] -> p1
   |((d,a)::q) ,((e,b)::r) when d<e -> (d,a)::(add q p2)
   |((d,a)::q), ((e,b)::r) when d>e -> add p2 p1
   |((d,a)::q), ((e,b)::r) when d=e -> if a+. b =0. then add q
```

```
r else (d,a+.b)::(add q r );;
```

3-

```
let rec prodmon (n,a) p =
  if a =0. then [] else match p with
    |[]->[]
   |(d,b)::q ->(d+n,a*.b)::(prodmon (n,a) q);;
let rec prod p1 p2 = match p1 with
   |[] ->  []
   |(d,a)::q -> add (prodmon (d,a) p2) ( prod q p2);;
```

4- on prend garde à ne pas dériver (0,...) en (-1,...) (monomes constants)

```
let rec derive = fun
   |[]-> []
   |((0,a)::q)-> derive q
   |((d,a)::q) -> (d-1,float_of_int(d)*.a)::(derive q);;
```

5- si les polynômes n'étaient pas creux on utiliserait une liste formée des coefficients et pour un polynome creux il y aurait beaucoup de zeros.

(*exo 8*)
si depile enleve le sommet et le renvoie :

```
let repousse p= let s= depile p in let t=depile p in
empile t (empile s p);;
```

si depile renvoie simplement la queue et sommet renvoie le sommet (comme dans le cours sur le type a list) alors :

```
let repousse p = let s = sommet p in
let q = depile p in let t = sommet q in let r = depile q in
empile t (empile s r);;
```

(*exo 9*)
1- 3 2 5 sqrt * + 6 /

```
2- let l =[Nb  3.; Nb 2.; Fn sqrt; Nb 5.;Op( prefix *.);Op
(prefix +.);
  Nb 6.; Op (prefix /.)];;
```

```
3- pile      expression
 [3]   2  5 sqrt* + 6 /
[3;2]    5  sqrt * + 6 /
[3;2;5] sqrt * + 6 /
[3;2; (sqrt 5) ]     * + 6 /    (on a depilé 5, calculé sqrt 5
puis empilé le resultat)
[3; 2*sqrt 5]  + 6/
[3+ sqrt 2 * 5]     6 /
[3+ sqrt 2 * 5;6]     /
```

[resultat]

4- cf poly


```
(* exo 10 : takewhile dropewhile *)

let rec takewhile p = fun
 (*[]->[]*)
| l when p (hd l) -> (hd l):: (takewhile p (tl l))
| _ -> [];;

let rec dropewhile p = fun
 []->[]
|l when p (hd l) -> (dropewhile p (tl l)
| l-> l;;
```

la signature de appl est ('a -> bool) ->'a list ->bool
elle est normalement constante égale à true.

s est la concaténation de takewhile p s et de dropewhile p s.

(* exo 11 : mult*)

sur la derniere ligne il faut ecrire a b au lieu de _ _

f a b calcule a*b selon la règle a*(2n)=(2a)*n et a*(2n+1)=(2a)*n+a. (b>0)

Preuve : tout se passe bien pour b=1, et a quelconque, puis si on suppose que le calcul se termine
et est correct jusqu'à l'entier b , alors pour b+1

 * soit b+1 est pair et dans ce cas on appelle f (2a) (b+1)/2 qui termine bien et
 est correcte puisque (b+1)/2 est compris entre 1 et b.

 *soit b+1 est impair et dans ce cas on appelle f (2a) (b)/2 qui termine bien et
 est correcte puisque b/2 est compris entre 1 et b.

signature int -> int -> int

le nombre d'additions correspond au nb de 1 moins 1 dans l'écriture binaire de b.

(* exo 12 : tranches de somme minimale*)

```
1) let rec somme t = fun
 i j when i=j -> t.(j)
|i j ->(somme t i (j-1))+t.(j);;

2) let tranche_min1 t n =
    let m= ref t.(0) in
```

```
    for i=0 to n do
          for j=i to n do
             m:= min (somme t i j) !m
          done;
    done;
!m;;
```

3) somme requiert j-i additions, donc la deuxième boucle en requiert (n-i)(n+1-i)/2, et
tranche min en reclame som k(k+1)/2=n(n+1)^2/6. Ce qui est coûteux.

Il faut n(n+1)/2 comparaisons.

4)
```
let rec somme_min t j n = match j with
      j when j=n -> t.(n)
     |j-> let s= somme_min t (j+1) n in min t.(j) (s+t.(j));;
```

5)
```
let rec tranche_min2 t n= let m=ref t.(n) in
       for j=n-1 downto 0 do
            m:= min !m (somme_min t j n)
       done;
   !m;;
```

6) cette fois il y a n-j additions (et comparaisons) dans la boucle, ce qui est mieux.

7) 1)On examine V.(i+1) tout seul, et s + V.(i+1).

7) 2) On fait varier i, en gardantsmin et s qu'on actualise à chaque étape. Il n'y a donc
qu'une boucle for.

8)
```
let tranche_min3 t n =
            let smin = ref t.(0) and s = ref t.(0)
            and  d = ref 0 and f = ref 0 in let k = ref 0
             in
                       for i=1 to n  do
                              if t.(i) < !s + t.(i) then

begin

s := t.(i);

k := i;
                                          end
                                      else s := !s + t.(i);
                              if !s < !smin then
                                          begin
                                             smin := !s;
                                             d := !k; f := i

end;
                                     done;
                       !d, !f , !smin;;
```

(*Exo 13 : mots de Lyndon *)
Rem: Ici on numérote de 1 à n!!

1)

```
        let inferieur u v = u < v;;
```

2)

```
let conjugue u i =
       let n = string_length u in
       let v = sub_string u 0 (i-1) and w=sub_string u (i-1)
(n-i+1)
          in w^v;;
```

3)
\begin{verbatim}

```
let lyndon u =
       let n = string_length u in
       let rec aux = fun
            0 -> true
           |k -> inferieur u (sub_string u k (n-k)) && aux (k-
1)
          in
       aux (n-1);;
```

La fonction aux est à récursivité terminale, mais l'utilisation de substring n'est pas satisfaisante.
En Caml il vaudrait mieux travailler avec des listes!

4)

```
let factorisation u =
            let n = string_length u in
            let rec initialise l = fun
                      k when k=0 -> l
                     |k -> initialise ((char_for_read u.[k-
1])::l) (k-1)
            in let ul = initialise [] n in
            let rec compacte = fun
                 [] -> []
                |[a] -> [a]
                        |(a::b::q) when a<b ->  (a^b)::
(compacte q)
                |(a::b::q) -> a::compacte (b::q)
            in let ll =  ref ul in let lll = ref (compacte !
ll) in
            while !ll <> !lll do
               ll := !lll ; lll := compacte !ll done;
            !ll;;
```

5)
(0,0,0,0,1),(0,0,0,1,1),(0,0,1,1,1);(0,1,0,1,1);(0,0,1,0,1);

(0,1,1,1,1)

6)

```
let rec insere_mot_lst u = fun
       [] -> [u]
      |(a::q) when u < a -> u::a::q
      |(a::q) when u = a -> a::q
         |(a::q) -> a::(insere_mot_lst u q);;
```

7)

```
let rec insere_lst_lst lst1 lst2 =
             match lst1 with
       [] -> lst2
      |(a::q) -> insere_lst_lst q (insere_mot_lst a lst2);;
```

8)}

```
let rec fusionne_listes = fun
    [] lst2 ->  []
   |lst1 [] ->  []
   |(a::q) (b::r) -> let la = fusionne_listes (a::q) r and
                         lb = fusionne_listes q r in
                     let lc = insere_lst_lst la lb in
                              if a<b then insere_mot_lst (a^b)
lc
                                     else lc;;
```

9) La fonction s'appelle g\'en\`ere dans la liste des signatures!

```
let nmax=9;;
let lynd = make_vect (nmax+1) [];;
let remplir_lynd n =
     lynd.(1) <- ["0";"1"];
     for i=2 to n do
           for j=1 to i-1 do
              lynd.(i) <- insere_lst_lst lynd.(i)
(fusionne_listes lynd.(j) lynd.(i-j))
                         done done;;
```