1.) LISTES.

1.1.) Généralités.

Les listes sont des suites homogènes de valeurs, entourées de crochets "[" et "]". Comme pour les tableaux, les éléments des listes sont séparés par un point virgule ";".

```
#let liste = [1;2;3;4;5;6];;
liste : int list = [1; 2; 3; 4; 5; 6]
```

Au contraire des tableaux, on n'accède pas directement à un élément d'une liste : il faut parcourir séquentiellement la liste pour atteindre l'élément recherché. En revanche, les listes peuvent grossir dynamiquement alors que les tableaux ont une taille déterminée fixée lors de leur construction.

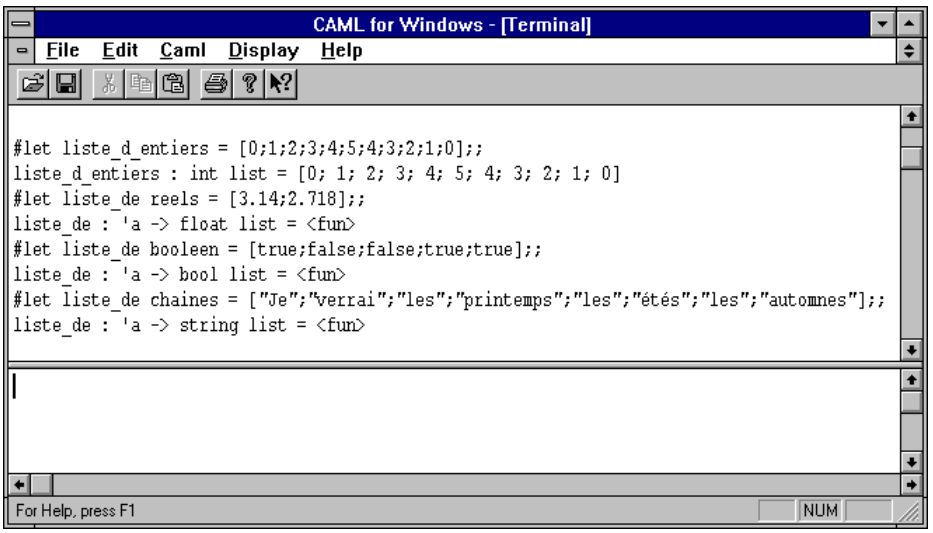


Figure 1 Listes

1.2.) Constructeurs.

Toutes les listes sont construites avec les deux constructeurs de listes "**[]**" (qu'on prononce ***nil***) et "**::**" (qu'on prononce "***conse***", par abréviation de constructeur).

```
#let liste = [];;
liste : 'a list = []
#3::liste;;
- : int list = [3]
#1::2::liste;;
- : int list = [1; 2]
```

"[]" est la liste vide et "::" est l'opérateur infixe qui ajoute un élément à une liste.

On peut accéder aux éléments d'une liste à l'aide des fonctions **hd** (head) qui renvoie le premier élément d'une liste et **tl** (tail) qui renvoie la liste sans le premier élément. La fonction list_length retourne quant à elle le nombre d'éléments de la liste.

`value hd : 'a list -> 'a`	Renvoie le premier élément d'une liste donnée. Renvoie une exception si la liste est vide.
`value tl : 'a list -> 'a list`	Renvoie la liste donnée en paramètre sans son premier élément. Déclenche une exception si la liste est vide.
`value list_length : 'a list -> int`	Retourne la longueur (nombre d'éléments) de la liste.

On pourrait par exemple, accéder à un élément d'une liste avec la fonction suivante :

```
#let rec element l n =
if n=0 then (hd l) else element (tl l) (n-1);;
element : 'a list -> int -> 'a = <fun>
```

1.3.) Filtrage des listes.

Le filtrage est étendu aux listes. On peut donc réécrire la fonction *élément* de la manière suivante :

```
#let rec element n = function
| a::r when n=0 -> a
| _::r -> element (n-1) r
| _ -> failwith "Element";;
'a = <fun>
element : int -> 'a list -> 'a = <fun>
```

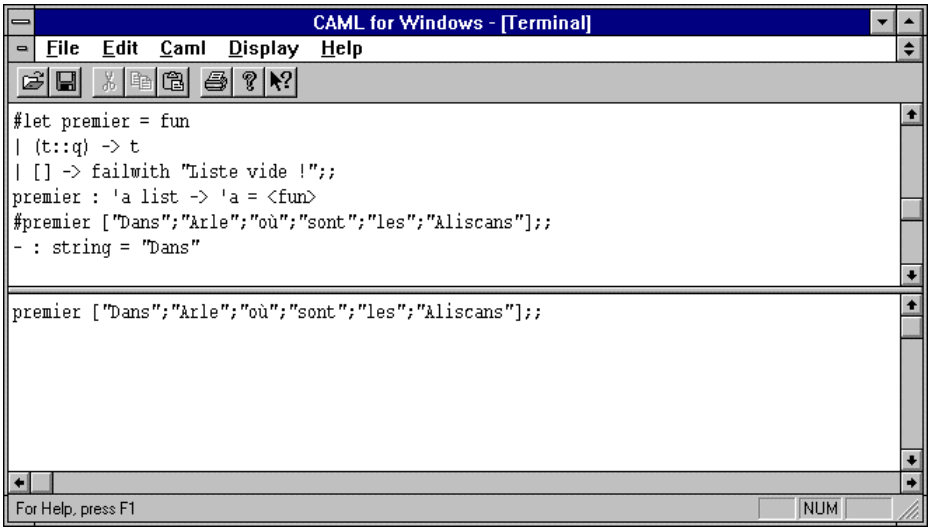


Figure 2 Filtrage de listes.

1.4.) Fonctions de manipulation de listes.

1.4.1.) CONCATENATION.

```
value prefix @ : 'a list -> 'a list -> 'a list
```

Exemple :

```
#[1;2;3] @ [4;5;6];;
- : int list = [1; 2; 3; 4; 5; 6]
```

1.4.2.) REVERSION.

```
value rev : 'a list -> 'a list
```

List reversal.

Exemple :

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

1.4.3.) EXISTENCE D'UN ELEMENT.

```
value mem : 'a -> 'a list -> bool
```
`mem a l` revoie vrai si et seulement si a est égal à un élément de la liste.

Exemple :

```
#mem (6/3) [1;2;3];;
- : bool = true
```

1.4.4.) SOUSTRACTION D'UN ELEMENT.

```
value except : 'a -> 'a list -> 'a list
```
`except a l` renvoie la liste l où le premier élément égal à a a été supprimé La liste l est renvoyée à l'identique si elle ne contient pas a.

Exemple :

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

1.4.5.) SOUSTRACTION DE LISTES.

```
value subtract : 'a list -> 'a list -> 'a list
```
`subtract l1 l2` retroune la liste l1 sans les éléments présents dans l2.

Exemple :

```
#subtract [1;2;3;4;5;6;7;8;9] [2;4;6];;
- : int list = [1; 3; 5; 7; 8; 9]
```

1.4.6.) UNION.

```
value union : 'a list -> 'a list -> 'a list
```
`union l1 l2` ajoute en tête de l2 tous les éléments de l1 non présents dans l2.

Exemple :

```
#union [2;4;6;8] [1;2;3;4;5];;
- : int list = [6; 8; 1; 2; 3; 4; 5]
```

1.4.7.) INTERSECTION.

`value intersect : 'a list -> 'a list -> 'a list`
`intersect l1 l2` retourne la liste des éléments communs à l1 et l2.

Exemple :

```
#intersect [2;4;6;8] [1;2;3;4;5];;
- : int list = [2; 4]
```

1.4.8.) INDEX D'UN ELEMENT.

`value index : 'a -> 'a list -> int`
`index a l` retourne la position du premier élément de l égal à a.

Exemple :

```
#index 3 [0;1;2;3;4;5;6;7;8;9];;
- : int = 3
```

1.5.) Fonctionnelles de listes.

Caml Light propose quelques fonctionnelles de listes :

1.5.1.) MAP.

`value map : ('a -> 'b) -> 'a list -> 'b list`

`map f [a1 ;...an]` applique la fonction f à chaque élément de `[a1 ;...an]` et construit la liste `[f a1 ;... f an]` avec les résultats de f.

Exemple :

```
#let f x = 2 * x;;
f : int -> int = <fun>
#map f [1;2;3];;
- : int list = [2; 4; 6]
```

1.5.2.) DO_LIST.

`value do_list : ('a -> 'b) -> 'a list -> unit`

`do_list f [a1 ;...an]` applique la fonction f à chaque élément de `[a1 ;...an]`. C'est équivalent à :

```
begin
     f a1;
     f a2;
      ...;
```

```
                f an;
                ()
        end.
```

Exemple :

```
#let f x = print_int x;print_newline();;
f : int -> unit = <fun>
#do_list f [1;2;3];;
1
2
3
- : unit = ()
```

1.5.3.) IT_LIST.

```
value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
```

`it_list f a [b1; ...; bn]` est équivalent à `f (... (f (f a b1) b2) ...) bn.`

Exemple :

```
#let f x y = x*y;;
f : int -> int -> int = <fun>
it_list f 3 [1;2;3];;
#- : int = 18
```

f (f (f 3 1) 2) 3 <=> f (f 3 2) 3 <=> f 6 3 <=> 18

1.5.4.) LIST_IT.

```
value list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
```

`list_it f [a1; ...; an] b` est équivalent à `f a1 (f a2 (... (f an b) ...))`.

Exemple :

```
#let f x y = x + y;;
f : int -> int -> int = <fun>
list_it f [1;2;3] 2;;
#- : int = 8
```

f (f 2 (f 3 2)) 1 <=> f (f 2 5) 1 <=> f 7 1 <=> 8

1.5.5.) MAP2.

```
value map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

`map2 f [a1; ...; an] [b1; ...; bn]` est équivalent à `[f a1 b1; ...; f an bn].` Cette fonctionnelle renvoie « `Invalid_argument "map2"` » si les deux listes ont des tailles différentes.

Exemple :

```
#let f x y = x + y;;
```

```
f : int -> int -> int = <fun>
map2 f [1;2;3] [4;5;6];;
#- : int list = [5; 7; 9]
```

1.5.6.) DO_LIST2.

`value do_list2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> unit`

`do_list2 f [a1; ...; an] [b1; ...; bn]` appelle `f a1 b1; ...; f an bn`, en ignorant le résultat de f. Cette fonctionnelle renvoie « `Invalid_argument "do_list2"` » si les deux listes n'ont pas la même longueur.

Exemple :

```
#let f x y = print_int(x + y);;
do_list2 f [1;2;3] [4;5;6];;
f : int -> int -> unit = <fun>
#579- : unit = ()
```

1.5.7.) IT_LIST2.

`value it_list2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a`

`it_list2 f a [b1; ...; bn] [c1; ...; cn]` est équivalent à `f (... (f (f a b1 c1) b2 c2) ...) bn cn`. Cette fonctionnelle renvoie « `Invalid_argument "it_list2"` » si les deux listes ont des longueurs différentes.

Exemple :

```
#let f x y z=  x*y + z;;
it_list2 f 2 [1;2;3] [4;5;6];;
f : int -> int -> int -> int = <fun>
#- : int = 57
```

`f (f (f 2 1 4) 2 5) 3 6 <=> f (f 6 2 5) 3 6 <=> f 17 3 6 <=> 57`

1.5.8.) LIST_IT2.

`value list_it2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c`

`list_it2 f [a1; ...; an] [b1; ...; bn] c` est équivalent à `f a1 b1 (f a2 b2 (... (f an bn c) ...))`. Cette fonctionnelle renvoie « `Invalid_argument "list_it2"` » si les deux listes ont des longueurs différentes.

Exemple :

```
#let f x y z=  x*y + z;;
f : int -> int -> int -> int = <fun>
list_it2 f [1;2;3] [4;5;6] 2;;
#- : int = 34
```

`f 1 4 (f 2 5 (f 3 6 2)) <=> f 1 4 (f 2 5 20) <=> f 1 4 30 <=> 34`

1.5.9.) FLAT_MAP.

```
value flat_map : ('a -> 'b list) -> 'a list -> 'b list
```
`flat_map f [l1; ...; ln]` est équivalent à `(f l1) @ (f l2) @ ... @ (f ln).`

Exemple :

```
let f x = let a = 2*x in [a];;
f : int -> int list = <fun>
flat_map f [1;2;3;4;5];;
#- : int list = [2; 4; 6; 8; 10]
```

1.5.10.) FOR_ALL.

```
value for_all : ('a -> bool) -> 'a list -> bool
```
`for_all p [a1; ...; an]` est équivalent à `(p a1) & (p a2) & ... & (p an).`

Exemple :

```
#let pair x = (x mod 2) = 0;;
pair : int -> bool = <fun>
#for_all pair [0;2;4;6;8;10];;
- : bool = true
#for_all pair [0;1;2;4;6;8];;
- : bool = false
```

1.5.11.) EXISTS.

```
value exists : ('a -> bool) -> 'a list -> bool
```
`exists p [a1; ...; an]` est équivalent à `(p a1) or (p a2) or ... or (p an).`

Exemple :

```
#let pair x = (x mod 2) = 0;;
pair : int -> bool = <fun>
exists pair [1;3;5;7;9;11];;
#- : bool = false
exists pair [1;3;4;5;7;9];;
#- : bool = true
```

1.6.) Exercices sur les listes.

1.6.1.) RECHERCHE DU PLUS PETIT ELEMENT D'UNE LISTE NON TRIEE.

```
let rec plus_petit = fun
| [] m -> m
| (t::q) m when t<m -> plus_petit q t
| (t::q) m -> plus_petit q m;;
```

Pour résoudre le problème de l'initialisation de *m* (*m* doit être le plus grand élément possible ou bien un élément de la liste) on pourra définir une deuxième fonction :

```
let le_plus_petit = fun
| [] -> failwith "Liste vide"
```

```
| (a::r) -> plus_petit r a;;
```

1.6.2.) SOMME DES ELEMENTS D'UNE LISTE

```
#let rec somme = function
| [] -> 0
| (r::q) -> r + somme q;;
somme : int list -> int = <fun>
```

1.6.3.) TEST DE PRESENCE D'UN ELEMENT DANS UNE LISTE

```
#let rec existe element = function
| [] -> false
| [a] when a=element -> true
| (t :: q) when t=element -> true
| (t :: q) -> existe element q ;;
existe : 'a -> 'a list -> bool = <fun>
#existe 1 [5;4;3;2;1;2;3;4;5];;
- : bool = true
#existe 6 [5;4;3;2;1;2;3;4;5];;
- : bool = false
```

1.6.4.) TRI PAR SELECTION

Voici une implémentation du tri par sélection dans une liste :

```
#let rec minimum_et_reste = function
| [x] -> (x,[])
| (t::q) -> let (m,r)= (minimum_et_reste q) in
            if t<m then (t,m::r) else (m,t::r)
| _ -> failwith "Erreur";;
minimum_et_reste : 'a list -> 'a * 'a list = <fun>
#let rec tri_par_selection = function
| [] -> []
| l -> let (m,r) = (minimum_et_reste l) in m::(tri_par_selection
r);;
tri_par_selection : 'a list -> 'a list = <fun>
#tri_par_selection [3;1;4;1;5;9;2;6;5;3;5;8;9;7];;
- : int list = [1; 1; 2; 3; 3; 4; 5; 5; 5; 6; 7; 8; 9; 9]
```

La fonction `minimum_et_reste` renvoie l'élément minimal d'une liste et la liste privée de cet élément minimal.
La fonction `tri_par_selection` place l'élément minimal de la liste en tête de liste puis recommence cette opération pour les éléments restants.

1.6.5.) TRI PAR INSERTION

Le tri par insertion peut se définir à l'aide de deux fonctions récursives, en utilisant le filtrage sur les listes :

```
#let rec tri = function
| [] -> []
| x :: l -> insert x (tri l)
and insert elem = function
| [] -> [elem]
| x :: l -> if elem < x then elem :: x :: l else x :: insert elem
l;;
tri : 'a list -> 'a list = <fun>
insert : 'a -> 'a list -> 'a list = <fun>
#tri ["Hugo";"Verlaine";"Baudelaire";"Villon"];;
- : string list = ["Baudelaire"; "Hugo"; "Verlaine"; "Villon"]
```

1.6.6.) TRI BULLE.

On pourrait définir le tri « bulle » sur des listes de la manière suivante :

```
#let rec une_passe = function
| [x] -> false,[x]
| (t::q) -> let inversion,res = (une_passe q) in
if t<=(hd res) then inversion,(t::res) else
true,(hd res)::t::(tl res)
| _ -> failwith "Erreur";;
une_passe : 'a list -> bool * 'a list = <fun>
let rec tri_bulle = function
| [] -> []
| l -> let (modifiee,liste) = une_passe l in
       if modifiee then (hd liste)::(tri_bulle (tl liste))
       else liste;;
tri_bulle : 'a list -> 'a list = <fun>
#tri_bulle [2;7;1;8;2;8;1;8;2;8;4;5;9;0];;
- : int list = [0; 1; 1; 2; 2; 2; 4; 5; 7; 8; 8; 8; 8; 9]
```

1.6.7.) PERMUTATIONS.

La fonction « `permutations` » renvoie toutes les permutations possibles des éléments d'une liste :

```
#let rec permut l = fun
| [] [] -> [l]
| debut [] -> []
| debut (t::q) -> (permut (l @ [t]) [] (debut @ q))
@ (permut l (debut @ [t]) q);;
permut : 'a list -> 'a list -> 'a list -> 'a list list = <fun>
#let permutations l =
permut [] [] l;;
permutations : 'a list -> 'a list list = <fun>
```

```
#permutations ["Marquise";"vos beaux yeux";"me font";"mourir"; "d'amour"];;
- : string list list =
  [["Marquise"; "vos beaux yeux"; "me font"; "d'amour"; "mourir"];
  ["Marquise"; "vos beaux yeux"; "me font"; "mourir"; "d'amour"];
  ["Marquise"; "vos beaux yeux"; "d'amour"; "me font"; "mourir"];
  ["vos beaux yeux"; "Marquise"; "me font"; "d'amour"; "mourir"];
  ["vos beaux yeux"; "Marquise"; "me font"; "mourir"; "d'amour"];
  ["me font"; "Marquise"; "vos beaux yeux"; "d'amour"; "mourir"];
  ["me font"; "Marquise"; "vos beaux yeux"; ...]; ...]
```

1.6.8.) PARTIES.

La fonction suivante renvoie toutes les parties d'une liste :

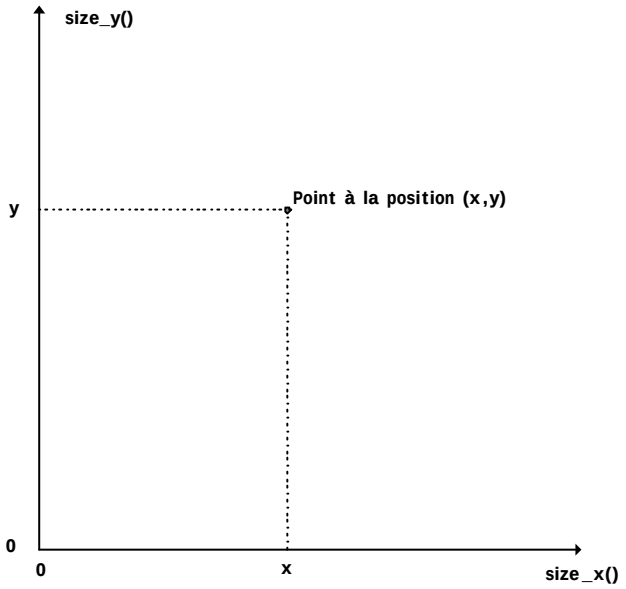
```
#let rec part l1 l2 =
if l1<>[] then part (tl l1) (hd l1::l2) @ part (tl l1) l2 else
[l2];;
part : 'a list -> 'a list -> 'a list list = <fun>
let parties liste = part liste [];;
parties : 'a list -> 'a list list = <fun>
#parties ["A";"B";"C"];;
- : string list list =
 [["C"; "B"; "A"]; ["B"; "A"]; ["C"; "A"]; ["A"]; ["C"; "B"]; ["B"];
  ["C"]; []]
```

2.) Graphismes en Caml.

L'accès aux fonctions graphiques se fait après ouverture de la bibliothèque graphique. Cette ouverture se fait par l'instruction

```
#open "graphics";;
```

2.1.) Coordonnées graphiques.



2.2.) Exceptions.

```
exception Graphic_failure of string.
```

Cette exception est renvoyée par les fonctions graphiques ci-après lorsqu'une erreur survient.

2.3.) Primitives graphiques.

2.3.1.) INITIALISATION.

```
value open_graph : string -> unit
```

Montre la fenêtre graphique ou bascule dans le mode graphique. Cette primitive efface la fenêtre graphique. La chaîne de caractère passée en paramètre est utilisée pour passer des informations optionnelles quant au mode graphique, la taille de la fenêtre etc... Son interprétration dépend de la machine.

```
value close_graph : unit -> unit
```

Ferme la fenêtre graphique et retourne en mode texte.

```
value clear_graph : unit -> unit
```

Efface le contenu de la fenêtre graphique.

```
value size_x : unit -> int
```

Donne la taille maximale de la fenêtre graphique (largeur).

```
value size_y : unit -> int
```

Retourne la taille maximale de la fenêtre graphique (hauteur).

2.3.2.) COULEURS

```
type color == int
```

Une couleur peut être spécifiée par ses composantes R, G, B (Rouge, Vert, Bleu). La valeur de chaque composante peut varier entre 0 et 255.

```
value rgb : int -> int -> int -> color
```

`rgb r g b` renvoie un entier représentant la couleur composée des trois valeurs r, g et b.

```
value set_color : color -> unit
```

Positionne la couleur de dessin courante.

```
value black : color
value white : color
value red : color
value green : color
value blue : color
value yellow : color
value cyan : color
value magenta : color
```

Couleurs prédéfinies.

```
value background : color
```

Couleur de fond par défaut.

```
value foreground : color
```

Couleur de dessin par défaut.

2.3.3.) POINTS ET LIGNES.

```
value plot : int -> int -> unit
```

`plot x y` dessine un point à la position `(x,y )` avec la couleur courante.

```
value point_color : int -> int -> color
```

`point_color x y` tetourne la couleur du point `(x,y)`.

```
value moveto : int -> int -> unit
```

`moveto x y` place le crayon à la position `(x,y)`.

```
value current_point : unit -> int * int
```

Renvoie la position du crayon.

```
value lineto : int -> int -> unit
```

`lineto x y` trace une ligne de la position courante jusqu'au point `(x,y)`.

```
value draw_arc : int -> int -> int -> int -> int -> int -> unit
```

`draw_arc x y rx ry a1 a2` trace un arc elliptique de `(x,y)`, de rayon horizontal `rx`, de rayon vertical `ry`, de l'angle `a1` à l'angle `a2` (en degrés).

```
value draw_ellipse : int -> int -> int -> int -> unit
```

`draw_ellipse x y rx ry` trace une ellipse de centre `(x,y)`, de rayon horizontal `rx` et de rayon vertical `ry`.

```
value draw_circle : int -> int -> int -> unit
```

`draw_circle x y r` trace un cercle de centre `(x,y)` et de rayon `r`.

```
value set_line_width : int -> unit
```

Fixe l'épaisseur du crayon.

2.3.4.) DESSIN DE TEXTES.

```
value draw_char : char -> unit
```

Trace un caractère à la position courante.

```
value draw_string : string -> unit
```

Trace une chaîne de caractères à la position courante.

```
value set_font : string -> unit
```

Fixe la police de caractères courante.

```
value set_text_size : int -> unit
```

Fixe la taille de la police de caractères courante.

```
value text_size : string -> int * int
```

Retourne la taille du texte passé en paramètre, si celui-ci est tracé avec la police courante.

2.3.5.) REMPLISSAGE DE ZONES.

```
value fill_rect : int -> int -> int -> int -> unit
```

`fill_rect x y w h` rempli le rectangle ayant le coin bas gauche à la position `(x,y)` de largeur `w` et de hauteur `h`, avec la couleur courante.

```
value fill_poly : (int * int) vect -> unit
```

Rempli un polygone avec la couleur courante. Le tableau passé en paramètre doit contenir les coordonnées de sommets du polygone.

```
value fill_arc : int -> int -> int -> int -> int -> int -> unit
```

Rempli un arc avec la couleur courante.

```
value fill_ellipse : int -> int -> int -> int -> unit
```

Rempli une ellipse avec la couleur courante.

```
value fill_circle : int -> int -> int -> unit
```

Rempli un cercle avec la couleur courante.

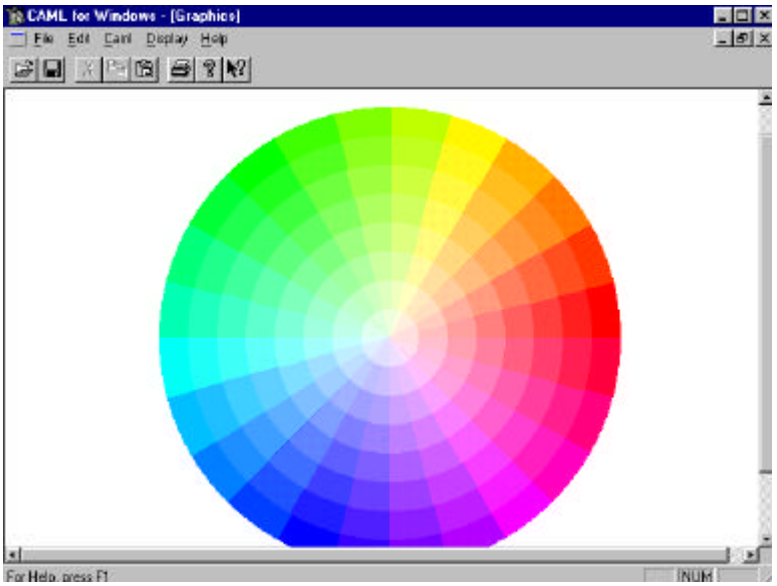


Figure 3 Exemple de graphisme en Caml[1].

2.4.) Quelques programmes graphiques.

2.4.1.) POLYGONES REGULIERS.

En plaçant k points également espacés sur un cercle, on obtient ce qu'on appelle un polygone régulier. Les 3 fonctions suivantes utilisent ce principe.

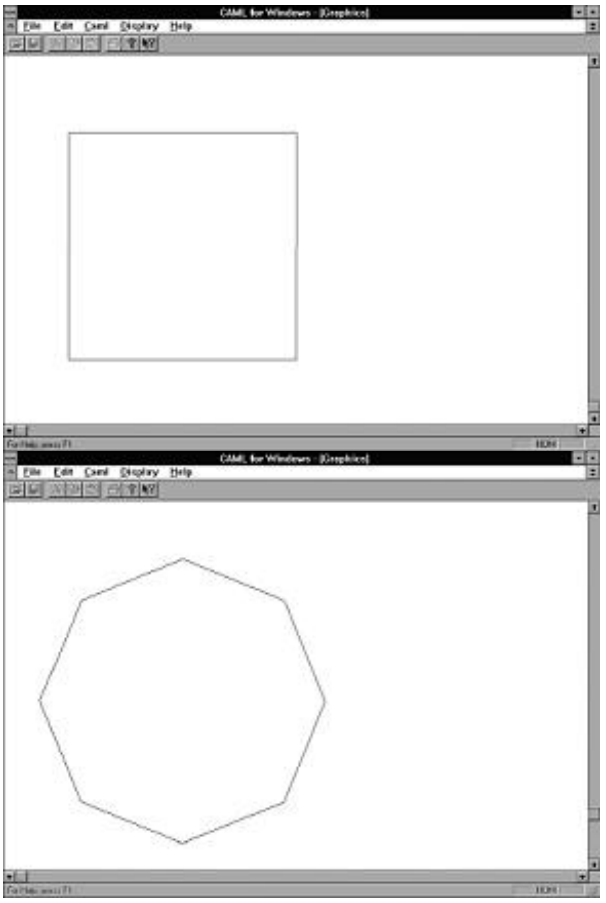
[1] Le code source de ce programme se trouve avec d'autres exemples Caml dans le répertoire `.../Caml/Examples/Colwheel/Colwhell.ml`.

```
let polygone k =
let np=480. and pi=3.14 in
let cx=np /. 2. (*Coordonnées du centre du cercle *)
and cy= np /. 2.
and r = np *. 0.45 (* rayon du cercle *)
and ad = pi /. 4. in
let x= ref 0. and y = ref 0. in
for i= 0 to k do
begin
   x:=cx  +.  r  *.  cos  (2.  *.  float_of_int  (i)  *.  pi  /.
float_of_int (k) +. ad);
   y:=  cy  +.  r  *.  sin(2.  *.  float_of_int  (i)  *.  pi  /.
float_of_int (k) +. ad);
   if i=0 then moveto (int_of_float !x) (int_of_float !y) else
   lineto (int_of_float !x) (int_of_float !y);
end;
done;;
```

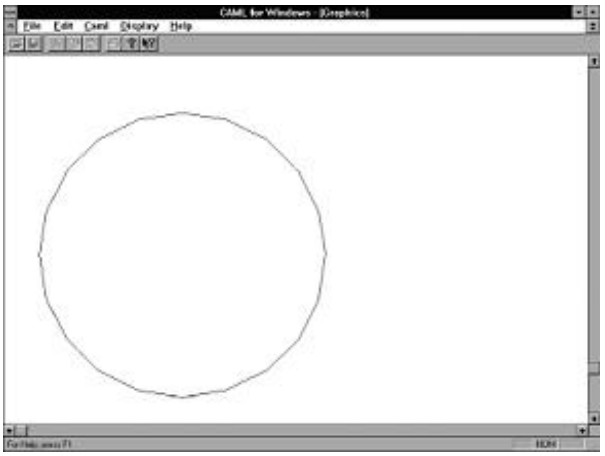
Voici quelques applications de cette fonction :

```
polygone 4;;
```

```
polygone 8;;
```



```
polygone 20;;
```



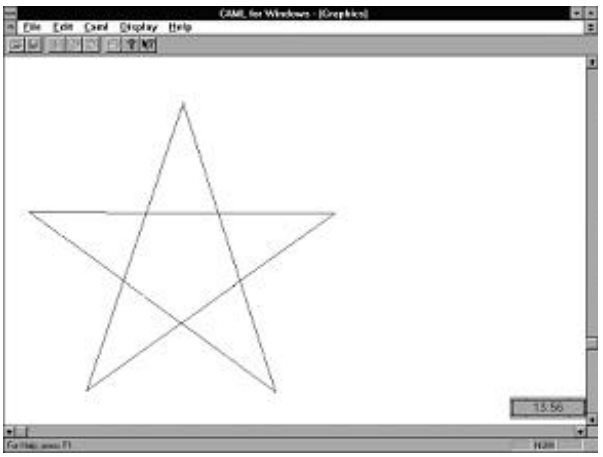
2.4.2.) ETOILES REGULIERES.

On considère à nouveau k points également espacés sur un cercle, mais cette fois ci, on joint ces points en sautant à chaque fois un même nombre h.

```
let etoiles k h =
let np=480.
and pi=3.14 in
let cx=np /. 2. (* Centre du cercle *)
and cy= np /. 2.
and r = np *. 0.45 (* Rayon du cercle *)
and ad = pi /. 2. in
let x= ref 0. and y = ref 0. in
for i= 0 to k do
begin
   x:=cx +. r *. cos (2. *. float_of_int (i) *. h *. pi /.
float_of_int (k) +. ad);
   y:= cy +. r *. sin(2. *. float_of_int (i) *. h *. pi /.
float_of_int (k) +. ad);
   if i=0 then moveto (int_of_float !x) (int_of_float !y) else
   lineto (int_of_float !x) (int_of_float !y);
end;
done;;
```

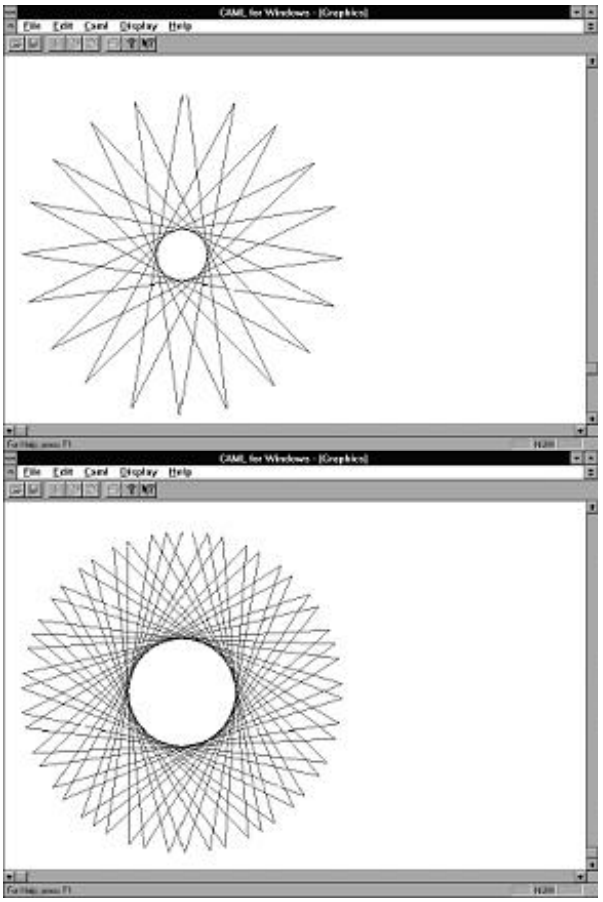
Voici quelques exemples :

```
etoiles 5 3;;
```



```
etoiles 20 9;;
```

```
etoiles 51 20;;
```

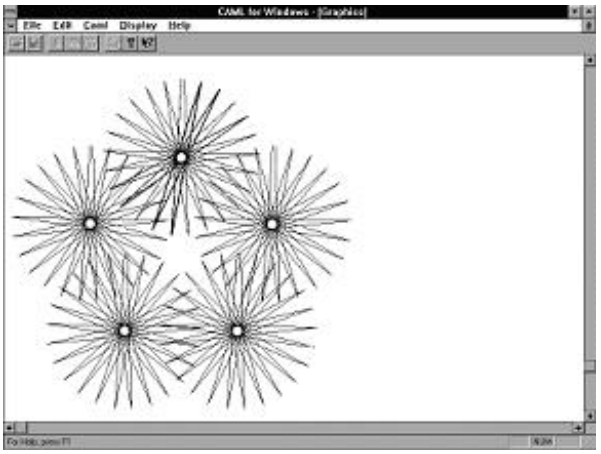


2.4.3.) COMPOSITIONS.

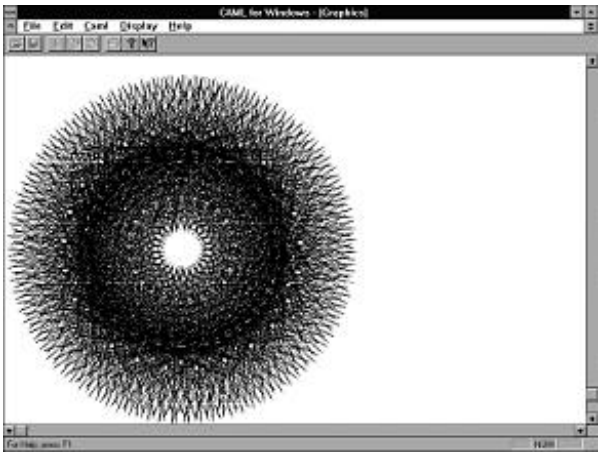
Dans cette fonction, on dessine k1 étoiles régulières dont les centres seront également espacées sur un cercle (autrement dit, dont les centres seront les sommets d'un polygone régulier).

```
let etoiles k h cx cy r ad =
let np=480.
and pi=3.14 in
let x= ref 0. and y = ref 0. in
for i= 0 to k do
begin
   x:=cx +. r *. cos (2. *. float_of_int (i) *. h *. pi /.
float_of_int (k) +. ad);
   y:= cy +. r *. sin(2. *. float_of_int (i) *. h *. pi /.
float_of_int (k) +. ad);
   if i=0 then moveto (int_of_float !x) (int_of_float !y) else
   lineto (int_of_float !x) (int_of_float !y);
end;
done;;
let composition k =
let np=480. and pi=3.14 in
let cx=np /. 2. (*Coordonnées du centre du cercle *)
and cy= np /. 2.
and r = np *. 0.27 (* rayon du cercle *)
and ad = pi /. 2. in
let x= ref 0. and y = ref 0. in
for i= 0 to k do
begin
   x:=cx +. r *. cos (2. *. float_of_int (i) *. pi /.
float_of_int (k) +. ad);
   y:= cy +. r *. sin(2. *. float_of_int (i) *. pi /.
float_of_int (k) +. ad);
   etoiles 25 12. !x !y (np *. 0.22) (pi /. 2.);
end;
done;;
```

```
composition 5;;
```

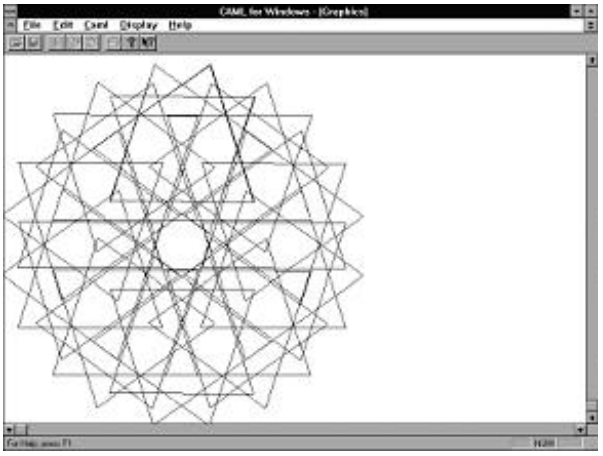


```
composition 50;;
```



En modifiant quelques paramètres, on obtient des effets très différents :

```
let composition k =
let np=480. and pi=3.14 in
let cx=np /. 2. (*Coordonnées du centre du cercle *)
and cy= np /. 2.
and r = np *. 0.27 (* rayon du cercle *)
and ad = pi /. 2. in
let x= ref 0. and y = ref 0. in
for i= 0 to k do
begin
   x:=cx  +.  r  *.  cos  (2.  *.  float_of_int  (i)  *.  pi  /.
float_of_int (k) +. ad);
   y:=  cy  +.  r  *.  sin(2.  *.  float_of_int  (i)  *.  pi  /.
float_of_int (k) +. ad);
   etoiles 10 3. !x !y (np *. 0.25) 0.;
end;
done;;
```

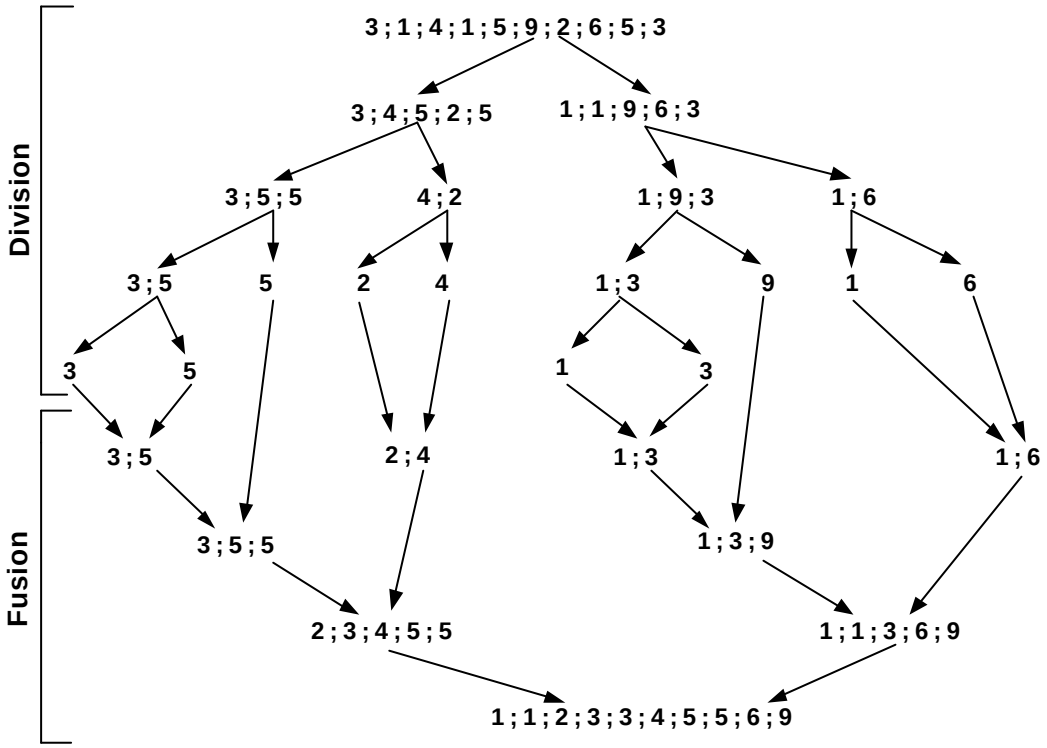


3.) Tri fusion (Mergesort).

Il s'agit à nouveau d'un tri suivant le paradigme *diviser pour régner*. Le principe en est le suivant :

- On divise en deux moitiés la liste à trier (en prenant par exemple, un élément sur deux pour chacune des listes).
- On trie chacune d'entre elles.
- On fusionne les deux moitiés obtenues pour reconstituer la liste triée.

Le tri fusion appliqué à la liste `[3;1;4;1;5;9;2;6;5;3]` se déroulera selon le schéma suivant :



On aura besoin pour cela de deux fonctions :

- Une fonction qui divise une liste en deux listes de longueur égale (ou presque dans le cas de listes de longueur impaires) :

```
#let rec divise = function
| [] -> ([],[])
| [un_element] -> ([un_element],[])
| (premier::second::reste) -> let (a,b) = divise reste in
                               (premier::a,second::b);;
divise : 'a list -> 'a list * 'a list = <fun>
#divise [1;2;3;4;5;6;7;8;9;0];;
- : int list * int list = [1; 3; 5; 7; 9], [2; 4; 6; 8; 0]
```

- Une fonction qui fusionne deux listes triées en une seule :

```
#let rec fusion = fun
| liste [] -> liste
| [] liste -> liste
| (a::b as liste_1) (c::d as liste_2) ->
                                    if a<c then a::(fusion b liste_2)
                                    else c::(fusion liste_1 d);;
fusion : 'a list -> 'a list -> 'a list = <fun>
```

La fonction `tri_fusion` se définie ensuite facilement :

```
#let rec tri_fusion = function
| [] -> []
| [e] -> [e]
| liste -> let (e1,e2) = divise liste in
fusion (tri_fusion e1) (tri_fusion e2);;
tri_fusion : 'a list -> 'a list = <fun>
#tri_fusion [3;1;4;1;5;9;2;6;5;3;5;8;9;7;9];;
- : int list = [1; 1; 2; 3; 3; 4; 5; 5; 5; 6; 7; 8; 9; 9; 9]
```

Pour les amateurs, voici une implémentation du tri « fusion » sur les vecteurs :

```
#let fusion v debut fin aux =
let m = (debut+fin)/2 and i = ref debut in let j = ref (m+1) in
  for k=0 to (fin-debut) do
    if (!i<=m) then
    begin
      if (!j <= fin) then
      begin
        if v.(!i) <= v.(!j) then
        begin
          aux.(k) <- v.(!i); incr i;
        end
        else
        begin
          aux.(k) <- v.(!j); incr j;
        end
      end
      else
      begin
        aux.(k)<-v.(!i);incr i
      end
    end
    else
    begin
      aux.(k) <- v.(!j);incr j;
    end
  done;
  for k=0 to (fin-debut) do
    v.(debut+k)<-aux.(k)
  done;;
fusion : 'a vect -> int -> int -> 'a vect -> unit = <fun>
let tri_fusion v =
let m = vect_length v in
```

```
let aux = make_vect m v.(0) in fusion v 0 (m-1) aux;;
tri_fusion : 'a vect -> unit = <fun>
fusion : 'a vect -> int -> int -> unit = <fun>
```

4.) Recherche de motifs dans une chaîne.

4.1.) Position du problème.

Nous nous intéressons ici à un problème très classique : On considère une chaîne ***Chaine*** de longueur L*g_chaine* et un motif *Motif* de longueur *Lg_motif*. On souhaite trouver, quand elle existe, la première occurrence de ***Motif*** dans la chaîne ***Chaine***, c'est à dire le plus petit entier i>= 0 tel que :

$$\forall k \in \{0,1,...Lg\_motif\}, Motif_{i+k} = Chaine_i$$

4.2.) Algorithme naï f.

Cet algorithme observe simplement qu'avec les notations précédentes (recherche d'un motif ***Motif*** de taille ***Lg_motif*** dans une chaîne ***Chaine*** de taille ***Lg_chaine***) l'occurrence éventuelle ***i*** de ***Motif*** dans ***Chaine*** vérifie bien entendu ***i+Lg_motif<Lg_chaine***.

On va donc successivement, pour ***i*** variant de 0 à ***Lg_chaine-Lg_motif***, comparer le motif ***Motif*** avec la sous-chaîne ***Chaine[i..i+(Lg_motif-1)]***.

4.2.1.) VERSION ITERATIVE.

```
#exception Echec and Reussite;;
L'exception Echec est définie.
L'exception Reussite est définie.

#let position motif chaine =
     let Lg_motif = string_length motif
     and Lg_chaine =string_length chaine
     and i=ref 0
     and k = ref 0
     in
     try while !i <= Lg_chaine-Lg_motif do
       k:=0;
       while !k < Lg_motif && motif.[!k] = chaine.[!i + !k] do
         incr k
       done;
       if !k = Lg_motif then raise Reussite
       else incr i;
     done;
     raise Echec;
     with Reussite -> !i;;
position : string -> string -> int = <fun>
```

4.2.2.) VERSION FONCTIONNELLE.

```
#exception Echec;;
L'exception Echec est définie.

#let position motif chaine =
     let Lg_motif = string_length motif
     and Lg_chaine = string_length chaine
     in
     let rec coïncide i j =
          motif.[j] = chaine.[i] &&
          (j=Lg_motif-1 || coïncide (i+1) (j+1))
     in
     let rec teste i =
          if i> Lg_chaine-Lg_motif then raise Echec
          else if coïncide i 0 then i
          else teste (i+1)
     in
     teste 0;;
position : string -> string -> int = <fun>
```

4.2.3.) EVALUATION.

La complexité au pire de cet algorithme est en O(nm).

4.3.) Algorithme de Knuth, Morris et Pratt.

4.3.1.) PRESENTATION DE L'ALGORITHME.

L'algorithme de Morris et Pratt repose sur une observation à propos de l'algorithme naïf. Supposons que l'on recherche le motif "*abacabac*" dans la chaîne "*abacacabacababaabbab*".

On cherche d'abord si le motif est en tête de la chaîne : pour $i = 0$, on teste les égalités $Chaine_{i+k}=Motif_k$, k variant de 0 à $Lg\_motif-1$. L'égalité est prise en défaut pour k=5 car $Chaine_5='c'$ alors que $Motif_5='b'$.

L'observation de Morris et Pratt est la suivante : on peut affirmer au moment de l'échec des comparaisons qu'on a quand même $Chaine_i = Motif_0 ... Chaine_{i+k-1} = Motif_{k-1}$. Dans l'algorithme naïf, on comparerait ensuite les égalités $Chaine_{1+k} = Motif_k$, pour k variant de 0 à $Lg\_Motif - 1$, ce qui revient à comparer un préfixe de $Motif$ à un suffixe de $Chaine[0..k-1]$.

On pourrait donc, au préalable, étudier la taille du plus grand préfixe égal à un suffixe de ces tronçons du motif. Dans, cet exemple, la partie commune à $Chaine$ et à $Motif$ est "**abaca**" et le plus grand préfixe égal à un suffixe de "**abaca**" est **a**. Ceci nous permet de poursuivre la recherche en positionnant le motif sous ce dernier **a**.

Chaîne	**a**	**b**	**a**	**c**	**a**	**c**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**b**
(1)	a	b	a	c	a	**b**	a	c												
(2)					a	**b**	a	c	a	b	a	c								
(3)						**a**	b	a	c	a	b	a	c							
(4)							a	b	a	c	a	b	a	**c**						
(5)											a	b	a	**c**	a	b	a	c		
(6)													a	b	a	**c**	a	b	a	c

Le motif M = **abacabac** est aligné avec la chaîne C = **abacacabacababaabbab** de telle sorte que les 5 premiers caractères coïncident. En nous servant que de notre connaissance des 5 premiers caractères concordants, on peut en déduire qu'un décalage de 1 caractère est invalide de même qu'un décalage de 2 ou 3 caractères. Par contre, un décalage de 4 caractères est cohérent avec tout ce qu'on connaît du texte et donc, potentiellement, valide. Les informations utiles pour ces déductions peuvent être calculées en comparant le motif avec lui-même. On voit en effet que le plus long préfixe de M également suffixe de M est « **abac** ».

La fonction « `calcule_bords` » exprime les correspondances entre le motif et ses propres décalages. La fonction `calcule_bords` renvoie un tableau T dont chaque élément $T.(q)$ est la longueur du plus grand préfixe de M qui est à la fois un suffixe de M_q

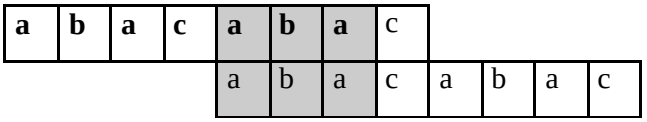
```
#let calcule_bords motif =
      let m = string_length motif
      in
      let r = make_vect (1+m) (-1)
      in
      let rec calcule j k =
           if k<0 || motif.[j-1] = motif.[k] then 1+k
           else calcule j r.(k)
      in
      for j=1 to m do r.(j) <- calcule j r.(j-1) done;
      r;;
calcule_bords : string -> int vect = <fun>

#calcule_bords  "abacabac";;
- : int vect = [|-1; 0; 0; 1; 0; 1; 2; 3; 4|]
```

Le résultat de la fonction « `calcule_bords` » peut s'interpréter ainsi :

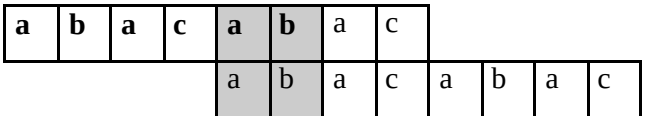
- T.(8) = 4 car le plus grand préfixe de M qui est un suffixe de M.[0]..M.[7] est 4 :

a	**b**	**a**	**c**	**a**	**b**	**a**	**c**				
				a	b	a	c	a	b	a	c

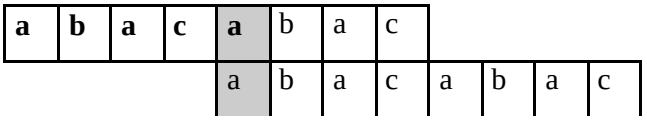
- T.(7) = 3 car le plus grand préfixe de M qui est un suffixe de M.[0]..M.[6] est 3 :



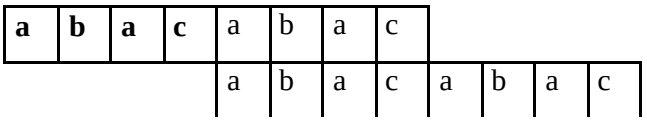
- T.(6) = 2 car le plus grand préfixe de M qui est un suffixe de M.[0]..M.[5] est 2 :



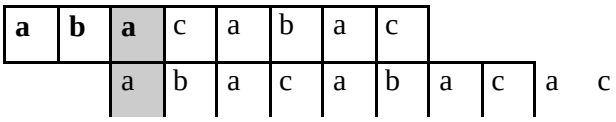
- T.(5) = 1 car le plus grand préfixe de M qui est un suffixe de M.[0]..M.[4] est 1 :



- T.(4) = 0 car il n'existe pas de préfixe de M qui soit suffixe de M.[0]..M.[3]:



- T.(3) = 1 car le plus grand préfixe de M qui est un suffixe de M.[0]..M.[3] est 1 :



Voici l'implémentation de l'algorithme :

```
#exception Echec;;
L'exception Echec est définie.

#let position motif chaine =
     let p = string_length motif
     and n = string_length chaine
     and i = ref 0
     and k = ref 0
     and r = calcule_bords motif
     in
     while !k < p && !i<n do
          if !k < 0 || chaine.[!i]=motif.[!k] then (incr i; incr k)
          else k := r.(!k)
     done;
     if !k >= p then !i-p
     else  raise Echec;;
position : string -> string -> int = <fun>
```

La fonction `position` commence par tester les caractères de Chaine concordant avec Motif. Ce test echoue pour Motif.(5).

Chaîne	**a**	**b**	**a**	**c**	**a**	**c**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**b**
(1)	a	b	a	c	a	**b**	a	c												

La question qui se pose alors est « quel est le plus grand préfixe de Motif correspondant à la partie déjà testée ? ». La réponse est « a » et la recherche reprend donc comme indiqué ci-dessous :

Chaîne	**a**	**b**	**a**	**c**	**a**	**c**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**b**
(1)	a	b	a	c	a	**b**	a	c												
(2)					a	**b**	a	c	a	b	a	c								

La recherche échoue avec Motif.[1]. Là encore on se pose la question de savoir quel est le plus grand préfixe de Motif correspondant à la partie déjà testée. L'algorithme continue ainsi jusqu'à ce que la chaîne ait été entièrement examinée.

Chaîne	**a**	**b**	**a**	**c**	**a**	**c**	**a**	**b**	**a**	**c**	**a**	**b**	**a**	**b**	**a**	**a**	**b**	**b**	**a**	**b**
(1)	a	b	a	c	a	**b**	a	c												
(2)					a	**b**	a	c	a	b	a	c								
(3)						**a**	b	a	c	a	b	a	c							
(4)							a	b	a	c	a	b	a	**c**						
(5)											a	b	a	**c**	a	b	a	c		
(6)													a	b	a	**c**	a	b	a	c

5.) Conception et stratégies algorithmiques.

5.1.) Les heuristiques.

Pour éviter l'explosion combinatoire lors de traitement complexes, on a introduit des règles dites heuristiques qui orientent la recherche dans une direction prometteuse au risque - dans certains cas - de ne pas conduire à la solution désirée.

Dans le langage informatique, une heuristique est une règle qui - sans être infaillible - conduit généralement à une situation à partir de laquelle une solution est plus facile.

5.2.) Les heuristiques gloutonnes.

Un algorithme glouton détermine une solution optimale pour un problème après avoir effectué une série de choix. Pour chaque point de décision de l'algorithme, le choix qui semble meilleur à cet instant est effectué. Cette stratégie ne produit pas toujours une solution optimale, mais c'est parfois le cas.

Dans un algorithme glouton, on fait le choix qui semble le meilleur sur le moment puis on résout les sous-problèmes qui surviennent une fois que le choix est fait.

Exemple d'algorithme glouton : décomposition d'une somme en billets de 500, 200, 100,50,20,10, 5 Euros et en pièces de 2 et de 1 Euros.

```
#let rec decompose n =
if (n / 500) > 0 then string_of_int (n/500) ^ " cinq cent, " ^
decompose (n mod 500)
else
if (n / 200) > 0 then string_of_int (n/200) ^ " deux cent, " ^
decompose (n mod 200)
else
if (n / 100) > 0 then string_of_int( n / 100) ^ " cent, " ^
decompose (n mod 100)
else
if (n / 50) > 0 then string_of_int (n / 50) ^ " cinquante, " ^
decompose (n mod 50)
else
if (n / 20) > 0 then string_of_int (n / 20) ^ " vingt, " ^ decompose
(n mod 20)
else
if (n/10) > 0 then string_of_int (n / 10) ^ " dix, " ^ decompose (n
mod 10)
else
if (n/2) > 0 then string_of_int(n/2) ^ " deux, " ^ decompose( n mod
2)
else string_of_int n ^ " un ";;
decompose : int -> string = <fun>
```

6.) Notions fondamentales à retenir.

6.1.) Langage Caml.

Définition: **`[]`**, `[ 1; 2; 3 ]` ou `x :: l` Accès:

```
match l with
| [] -> ...
| x :: l -> ...
```

Affectation: une liste est immuable.
Parcours: `do_list`, `map`
Fonctions: `list_length`

6.2.) Algorithmique.

6.2.1.) Le tri fusion.

6.2.2.) Les heuristiques gloutonnes.

7.) Exercices.

7.1.) Exercices.

EXERCICES 7.1.

Ecrivez une fonction qui, partant d'un texte, le transforme en une suite de chiffres en utilisant les règles suivantes :

- La ponctuation est ignorée.
- Chaque mot devient un chiffre égal au nombre de lettres du mot.

Par exemple, si on applique cette fonction au texte suivant :

Que j'aime à faire apprendre ce nombre utile aux sages
Immortel Archimède, artiste ingénieux
Qui de ton jugement peut priser la valeur ?
Pour moi, ton problème eut de pareils avantages.

on obtiendra : 3141592653589793238462643383279, c'est à dire les 3 suivi des 30 premières décimales de p . Le résultat sera donné sous forme d'une liste.

EXERCICES 7.2.

Les neufs premiers nombres peuvent être disposés dans un carré magique tel que les colonnes, les rangées et les diagonales présentent la même somme, égale à 15. Par exemple :

4	9	2
3	5	7
8	1	6

Ecrivez une fonction donnant tous les carrés magiques d'ordre 3 (3 lignes, 3 colonnes) pouvant être créés avec les nombres 1 ;2 ;3 ;4 ;5 ;6 ;7 ;8 ;9.

EXERCICES 7.2.

Traduisez en Caml le programme C suivant :

```
/* ------------------------------------------------ */
/* Recherche dichotomique dans un tableau trié      */
/* Retourne l'indice recherché et -1 en cas d'erreur */
/* ------------------------------------------------ */
int dicho (x , t, n )    /* Signature de la fonction */
int t[];         /* Tableau dans lequel on recherche */
int x;                          /* Elément recherché */
int n;               /* Nombre d'éléments du tableau */
  {
    int gauche, droite, milieu;
    gauche = 0;
```

```
    droite = n - 1;
    while (gauche < droite)
    {
         milieu = (gauche + droite) / 2;
         if (x <= t[milieu]) droite = milieu
         else gauche = milieu + 1;
     }
     return (x == t[gauche] ? gauche : -1);
  }
/* ------------------------------------------------ */
```

- Les caractères placés entre «`/*`» et «`*/`» sont des commentaires et ne sont pas pris en compte.
- Les trois lignes qui suivent la signature de la fonction permettent de préciser le type d'arguments attendus.
- En C, les accolades ouvertes «`{`» et fermantes «`}`» sont l'équivalent des `begin..end` en Caml.
- L'opérateur «`==`» est l'opérateur de test d'égalité. Sont équivalent Caml est «`=`».
- L'instruction «(*`condition ? expression1 : expression2`*)» à la fin de la fonction est un opérateur ternaire. Elle a pour équivalent Caml : «***`if condition then expression1 else expression2`***».
- Quand on affecte à un entier le résultat d'une division («`/`»), c'est de la division entière qu'il s'agit.
- La ligne «`int gauche, droite, milieu;` » indique la création de trois variables entières.
- Le mot clé «`return`» indique la valeur que la fonction doit renvoyer.

Si possible, essayez de proposer une version récursive.

EXERCICE 7.4

Traduction de chiffres romains en chiffres arabes. Dans un premier temps, on ne tiendra pas compte de la simplification consistant à écrire 4 "IV", 9 "IX" ou 900 "CM" (cette convention ne s'est d'ailleurs répandue qu'avec l'invention de l'imprimerie). Par exemple, 1999 s'écrira "MDCCCCLXXXXVIIII". Dans un deuxième temps, on tiendra compte de cette simplification. (Par exemple "MCMXCIX" = 1999.)