1.) Fonctionnelle et polymorphisme.

1.1.) Polymorphisme.

1.1.1.) DEFINITION.

Etymologiquement, *polymorphe* signifie plusieurs (*poly*) formes (*morphes*). On emploie ce terme en informatique pour désigner des objets ou des programmes qui peuvent servir sans modifications dans des contextes très divers. Par exemple, une fonction de tri sera monomorphe si elle ne s'applique qu'à un seul type de données (par exemple les entiers) et polymorphe si elle est capable de trier plusieurs types d'objets (par exemple des flottants, des chaînes de caractères ou des dates).

Le polymorphisme n'est pas l'apanage des fonctions : certaines valeurs non fonctionnelles peuvent aussi être utilisées avec plusieurs « formes », c'est à dire plusieurs types. Par exemple, si un vecteur d'entiers ne pourra pas être employé avec un autre type, un vecteur vide pourra quant à lui être employé avec un autre type.

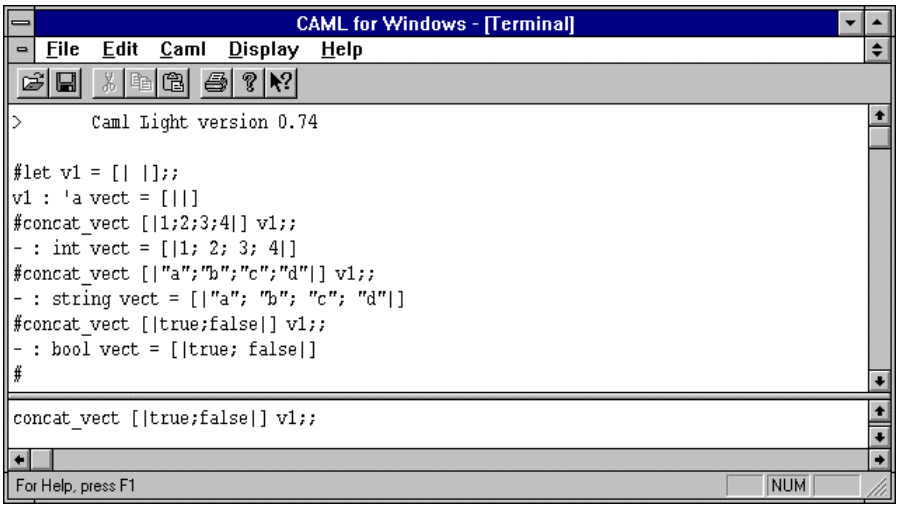


Figure 1 Un vecteur polymorphe.

1.1.2.) COMMENT INTRODUIT-ON LE POLYMORPHISME ?

Le polymorphisme consiste à introduire des schémas de type, c'est-à-dire des types avec des variables de type quantifiées universellement. Ces paramètres de schéma de type sont dénotés par le préfixe `'` (par exemple `'a`). Tous les paramètres des schémas de type sont quantifiés en tête du schéma: ainsi `'a -> 'a` signifie: « Pour tout type `'a`, `'a -> 'a` ». Cette quantification des paramètres de type est implicite en Caml:

```
#let identite a = a;;
identite : 'a -> 'a = <fun>
```

Le schéma de type de `identite` signifie Pour tout type `'a` la fonction retourne un type `'a`. On voit que la fonction peut s'appliquer aussi bien à des entiers, des booléens, des chaînes de caractères ou même des n-upplets.

```
#identite 5;;
- : int = 5
#identite true;;
- : bool = true
#identite "essai";;
- : string = "essai"
#identite (5,true,"essai");;
- : int * bool * string = 5, true, "essai"
```

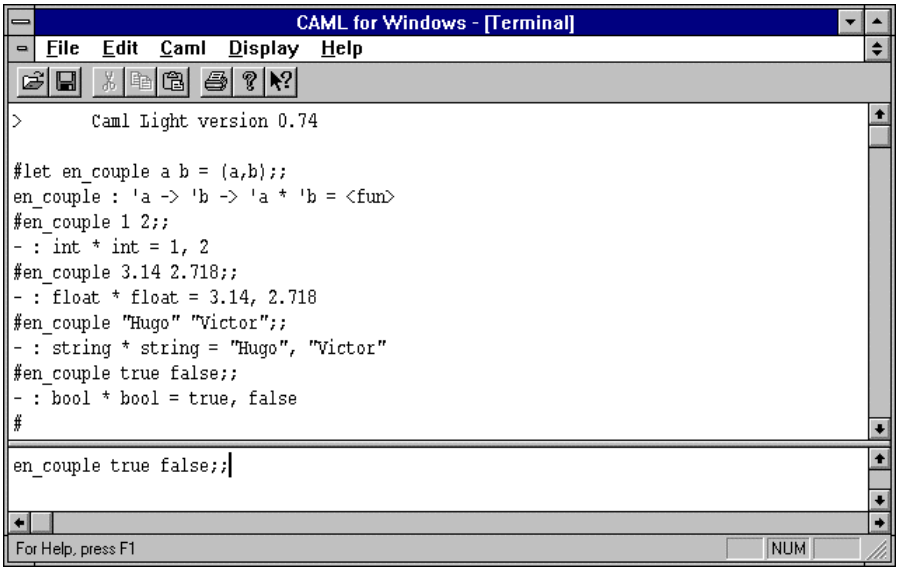


Figure 2 Une fonction polymorphe.

1.2.) Fonctions d'ordre supérieur.

Les fonctions d'ordre supérieur (ou « fonctionnelles ») sont des fonctions dont les arguments ou les résultats sont des fonctions.

1.2.1.) FONCTIONS RETOURNANT DES FONCTIONS.

Dans l'exemple suivant :

- On définit la fonction f qui additionne ses deux paramètres.
- On définit la fonction g qui est égale à l'application partielle de f.
- On applique g.

```
#let f x y = x + y;;
f : int -> int -> int = <fun>
#let g = f 1;;
g : int -> int = <fun>
#g 3;;
- : int = 4
```

On a défini la fonction g non pas en donnant les arguments et le corps de la fonction (par la construction function) mais par un calcul.

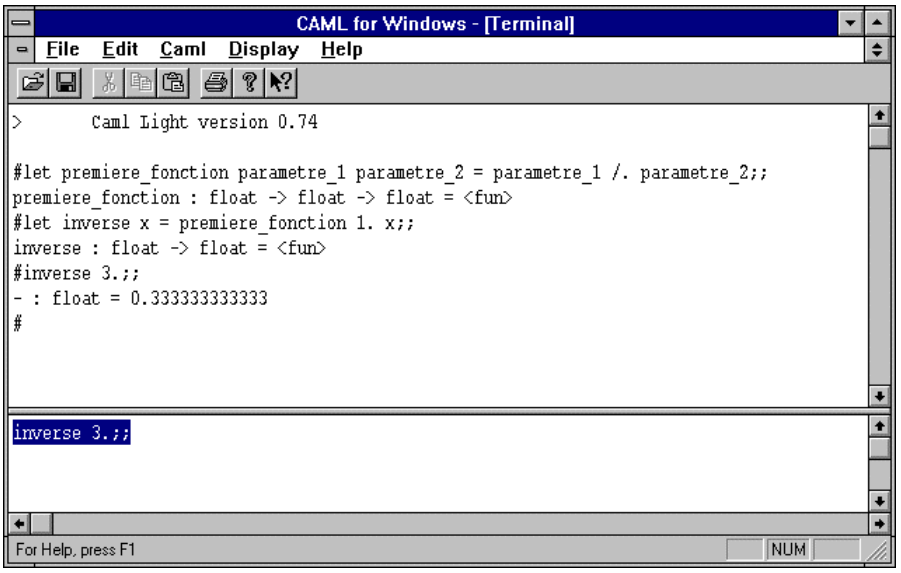


Figure 3 : Application partielle de fonctions.

`let fois_x (x) = (function y->x*y);;` `fois_x : int -> int -> int = <fun>`	On défini une fonction qui, étant donné un entier x, fabrique la fonction qui multipliera par x.
`#let double = fois_x 2;;` `#double : int -> int = <fun>`	On obtient la fonction qui multiplie par 2 en appliquant fois_x à 2.
`#double 4;;` `- : int = 8`	

1.3.) Fonctions dont les arguments sont des fonctions.

`let sigma formule n =`
`let resultat = ref 0 in`
`for i=0 to n do resultat := !resultat + formule(i) done;`
`!resultat;;`
`sigma : (int -> int) -> int -> int = <fun>`
	On défini une fonction qui accepte une autre fonction en argument.

`#sigma (function i->i*2) 5;;` `- : int = 30`	
`#sigma double 5;;` `- : int = 30`	

`#let compose g f x = g ( f x);;` `compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>`	
`#let f x = x + 1;;` `f : int -> int = <fun>`	
`#let g x = x*x;;` `g : int -> int = <fun>`	
`#let let h = compose g f;;` `h : int -> int = <fun>`	
`#h 3;;` `- : int = 16`	

On a défini h comme la composée g o f, c'est à dire l'application qui a n associe $(n+1)^2$. Il est possible de noter o l'opérateur binaire et de le déclarer infixe (c'est à dire placé entre ses arguments) à l'aide de la directive Caml #infix.

1.4.) Exemples de fonctionnelles existantes dans Caml.

`value map_vect : ('a -> 'b) -> 'a vect -> 'b vect`	map_vect f v applique la fonction f à chaque élément de v et construit un vecteur avec les résultats de f.
`value do_vect : ('a -> 'b) -> 'a vect -> unit`	do_vect f v applique la fonction f à tous les éléments de f.

1.5.) Exemples d'applications des fonctionnelles.

1.5.1.) CODAGE ET DECODAGE.

Ecrire une fonction qui code et décode une chaîne de caractère en utilisant le principe suivant :

- Remplacer chaque caractère de la chaîne entre A et Y par son suivant dans l'ordre alphabétique.
- Remplacer le caractère Z par A.

```
#let plus = function
| c when (c >=`A`) && (c <= `Y`) -> let i = int_of_char c in
char_of_int (i+1)
| `Y` -> `A`
| c when (c >=`a`) && (c <= `y`) -> let i = int_of_char c in
char_of_int (i+1)
| `y` -> `y`
| autre -> autre;;
plus : char -> char = <fun>
#let moins = function
| c when (c >=`B`) && (c <= `Z`) -> let i = int_of_char c in
char_of_int (i-1)
| `A` -> `Z`
| c when (c >=`b`) && (c <= `z`) -> let i = int_of_char c in
char_of_int (i-1)
| `a` -> `z`
| autre -> autre;;
moins : char -> char = <fun>
```

```
#let code s codage =
for i=0 to (string_length s -1) do
     set_nth_char s i (codage s.[i]);
done;
s;;
code : string -> (char -> char) -> string = <fun
#code "A la fin tu es las de ce monde ancien" plus;;
- : string = "B mb gjo uv ft mbt ef df npoef bodjfo"
#code "B mb gjo uv ft mbt ef df npoef bodjfo" moins;;
- : string = "A la fin tu es las de ce monde ancien"
```

1.5.2.) Recherche de racines par l'algorithme de Newton.

L'algorithme de Newton/Raphson consiste à approcher une solution de l'équation

$$f(x)=0$$

par la récurrence

$f(x_{n+1}) = f(x_n) - \frac{f(x_n)}{f'(x_n)}$ en prenant comme valeur initiale $f(x_n)$suffisamment proche de la racine.

La fonction `newton` ci-après reçoit deux fonctions en paramètres : la fonction dont on cherche la racine et sa dérivée.

```
#let rec newton f fp n =
if n=1 then 1. else
let x = newton f fp (n-1) in
x -. ((f x) /. (fp x));;
newton : (float -> float) -> (float -> float) -> int -> float =
<fun>
```

Par exemple, si on cherche une racine de l'équation `x3-2=0` on définira deux fonctions :

```
#let fonction x = x *. x *. x -. 2.;;
fonction : float -> float = <fun>
#let derivee x = 2. *. x *. x;;
derivee : float -> float = <fun>
```

Puis on appliquera la fonction `newton` :

```
#newton fonction derivee 30;;
- : float = 1.25992105043
```

On aurait aussi pu utiliser les fonctions anonymes :

```
#newton (function x -> x *. x *. x -. 2.) (function x -> 2. *. x *.
x) 30;;
- : float = 1.25992105043
```

1.5.3.) RECHERCHE DE RACINES PAR DICHOTOMIE.

La recherche d'une racine par dichotomie consiste à approcher

La fonction `dicho` ci-après reçoit en paramètres : l'intervalle dans lequel on recherche la racine, la précision demandée et la fonction dont on cherche la racine.

```
#let dicho a b p f =
let ra = ref a and rb = ref b and x = ref 0. in
while (!rb -. !ra) > p do
     x:= (!ra +. !rb) /. 2.;
     if f(!ra) *. f(!x) < 0. then rb := !x else ra := !x;
done;
!x;;
dicho : float -> float -> float -> (float -> float) -> float = <fun>
```

Par exemple, si on cherche une racine de l'équation `x3-2=0` on écrira :

```
#dicho 0. 3. 1e-15 (function x -> x *. x *. x -. 2.);;
- : float = 1.25992104989
```

2.) Conception et stratégies algorithmiques.

2.1.) Conception "diviser pour résoudre".

2.1.1.) TERMINOLOGIE.

La stratégie « diviser pour résoudre[1] » consiste à découper la donnée que l'on doit traiter en deux parts (ou plus) à peu près égales, puis à appliquer une (ou plusieurs) fois l'algorithme à l'une au moins de ces parts, avant de combiner les résultats obtenus pour construire le résultat correspondant à la donnée initiale.

Si le partage se fait le plus équitablement possible, une donnée de taille n va donc être divisée en deux données de taille [n/2]. Si on note f(n) le coût du partage et de la recombinaison, le coût c(n) vérifie :

$$c(n) = ac(n/2)+bc(n/2)+f(n)$$

a et b étant deux constantes entières non simultanément nulles. Dans la plupart des cas, a=b=1 (chaque moitié se voit appliquer une fois l'algorithme) ou a+b=1 (une et une seule moitié se voit appliquer l'algorithme).

2.2.) Applications.

2.2.1.) FONCTION PUISSANCE.

Prenons l'exemple de la fonction puissance :

```
let rec puissance a = function
| 0->1
| n->a*puissance a (n-1) ; ;
```

L'algorithme d'exponentiation binaire part du constat simple suivant : si, par exemple, on doit calculer a^{11} alors, plutôt que de faire 10 multiplications `a*a*a...*a`, on peut avantageusement calculer `b=a*a`, puis `c=b*b` puis `d=c*c` et finalement faire le produit `a*b*d`.

Ceci est bien plus économique en nombre de multiplications. Le principe est le suivant : si on écrit 11 en base 2 soit $b_3b_2b_1b_0$=1011, on a calculé les trois premiers termes de la suite $\left(a^{2^k}\right)\ k \geq 1$, puis multiplié entre eux les carrés correspondants aux b_i égaux à 1.

Dans le cas général, on effectue donc $\log_2 n$ multiplications pour le calcul des termes de la suite des carrés et «le nombre de 1 intervenants dans l'écriture binaire de n» pour les produits finaux. Ce deuxième entier est majoré par $\log_{(2)} n$. Donc, de linéaire, le nombre de multiplications est devenu logarithmique.

[1] En anglais : *divide and conquer*.

Le programme Caml, version itérative, correspondant est le suivant :

```
let puissance a n =
     let r= ref 1 and base2 = ref n and carre = ref a in
     while (!base2 > 0) do
          if (!base2 mod 2 = 1) then r := !r * !carre;
          base2 := !base2 / 2;
          carre := !carre* !carre ;
     done;
!r;;
```

Si on déroule le programme avec a=5 et n=15 on aura l'évolution des variables suivantes :

Itération	base2	base2 mod 1	r	carre
Début	15		1	5
1	7		25	25
2	3		125	625
3	1		78125	390625
4	0			

La version récursive :

```
let rec puissance a = function
| 0->1
| n -> let r = puissance a (n/2) in
if (n mod 2) = 0 then r*r else a*r*r;;
```

2.2.2.) TRI RAPIDE.

Le tri rapide[2] présenté ici est certainement l'algorithme de tri interne le plus efficace. Le principe de ce tri est d'ordonner le vecteur T.(0)..T.(n) en cherchant dans celui-ci une clé *pivot* autour de laquelle réorganiser ses éléments. Il est souhaitable que le *pivot* soit aussi proche que possible de la clé relative à l'enregistrement central du vecteur, afin qu'il y ait à peu près autant d'éléments le précédant que le suivant, soit environ la moitié des éléments du tableau. On permute ceux-ci de façon à ce que pour un indice *j* particulier tous les éléments dont la clé est inférieure à *pivot* se trouvent dans T.(0)...T.(j) et tous ceux dont la clé est supérieure se trouvent dans T.(j+1)...T.(n). On applique ensuite le tri récursivement à T.(1)..T.(1) et T.(j+1)..T(n) afin de trier ces deux segments de tableau. Comme tous les éléments du premier segment ont une clé inférieure ou égale à ceux du second, le tableau sera ainsi entièrement trié.

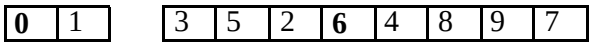
Supposons qu'on veuille trier le vecteur suivant : [|9;2;3;5;1;6;4;8;0;7|].

9	2	3	5	**1**	6	4	8	0	7

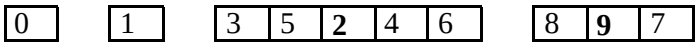
La clé pivot sera T.(4) c'est à dire «1».

Le vecteur sera divisé en deux parties. A gauche on place les éléments inférieurs ou égaux au pivot (T.(4) = 1) et à droite ceux supérieurs.

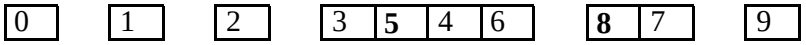
[2]Le « tri rapide » est encore appelé « tri de Hoare » (du nom de son inventeur) ou « tri par segmentation » ou « tri des bijoutiers » ou, en anglais, « quicksort ».



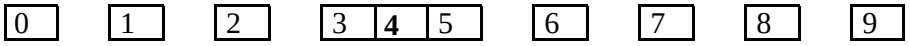
Dans chacune de ces 2 parties, on choisira un pivot. T.(0) c'est à dire «0» dans la première et T.(5) c'est à dire «6» dans la seconde. Chacune de ces 2 parties sera ensuite elle-même divisée en 2 parties, l'une contenant les éléments inférieurs au pivot et l'autre les éléments supérieurs :



Les deux premières parties ne contenant qu'un seul élément, elle sont considérées comme triées. Dans chacune des deux parties restantes on choisit un pivot. T.(4) c'est à dire «2» dans la première et T.(8) c'est à dire «9» dans la seconde. Chacune de ces deux parties sera ensuite divisée en 2 :



Il reste à nouveau 2 parties de plus de 1 élément. On choisira comme pivot T.(5) c'est à dire «5» et T.(8) c'est à dire «8». Puis on divise à nouveau :



Il reste 1 partie de plus de 1 élément. Le pivot choisi sera T.(4) c'est à dire «4».

Il reste 1 partie de plus de 1 élément. Le pivot choisi sera T.(3) c'est à dire «3».

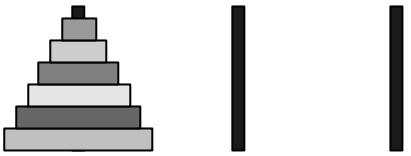
Il ne reste plus de parties de plus de 1 élément. Le vecteur est trié.

```
#let rec separer g d tab =
(* on sépare tab[g..d], en mettant dans la partie gauche de ce
vecteur les éléments de la suite qui sont <= pivot, et dans la
partie droite ceux qui sont > pivot, pivot étant un nombre a priori
quelconque *)
let pivot = tab.((g+d) / 2) and i = ref g and j = ref d in
while (!i < !j) do
    while (tab.(!i) < pivot) do i:=!i+1;done;
    while (tab.(!j) >pivot) do j:=!j-1;done;
    if (!i <= !j) then
    begin
         echange !i !j;
         i:=!i+1;j:=!j-1;
    end;
done;
if (g < !j) then separer g !j tab else ();
if (!i < d) then separer !i d tab else ();
where echange x y = let temp = tab.(x) in
tab.(x)<-tab.(y);tab.(y)<-temp;;
separer : int -> int -> 'a vect -> unit = <fun>
```

Dans le tri rapide, le choix du pivot est essentiel. Dans l'implémentation ci-dessus, on se contente de prendre l'élément situé au milieu de la partie à diviser. Mais d'autres stratégies sont possibles.

2.2.3.) TOURS DE HANOÏ.

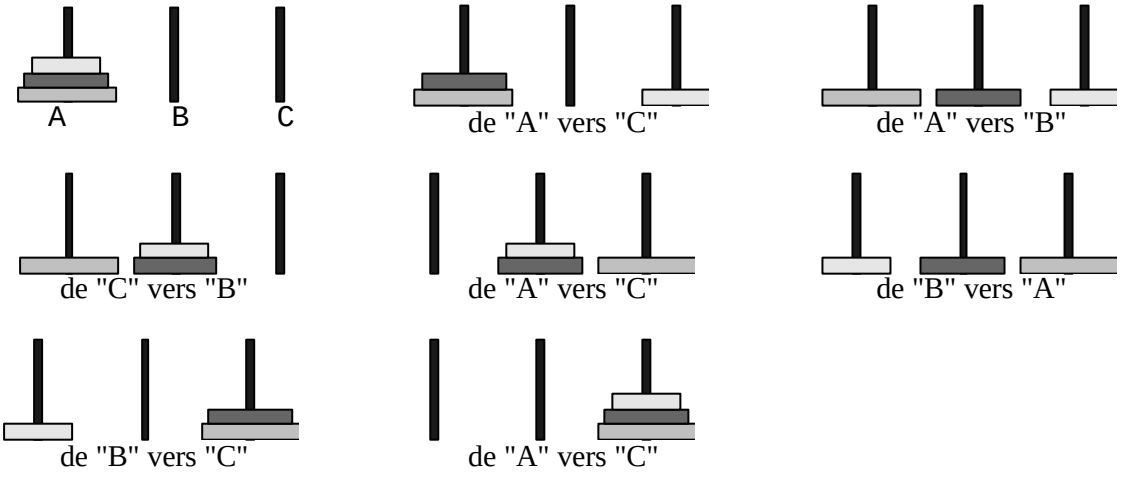
Le jeu des tours de Hanoï consiste en une plaquette sur laquelle sont fixées trois tiges. Sur ces tiges sont enfilés des disques de bois dont les diamètres sont tous différents :



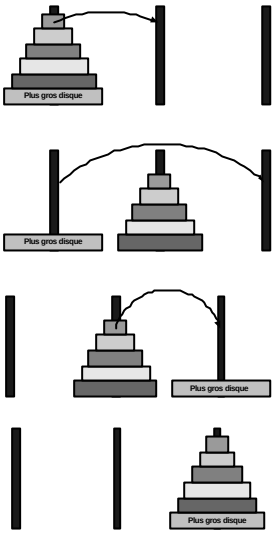
Au début du jeu, les disques sont empilés sur la tige de gauche. Le but du jeu est de déplacer les disques de la tige de gauche sur la tige de droite sans jamais violer les deux seules règles du jeu :

- On ne peut déplacer qu'un seul disque à la fois.
- On ne peut jamais poser un disque sur un disque plus petit.

Exemple avec 3 disques :



On suppose que les tiges s'appellent A, B et C et que n soit le nombre de disques qui sont posés sur la tige A et qu'il faut placer sur la tige C. L'astuce consiste à se rendre compte que si on sait résoudre le problème pour *n-1* disques, alors on sait le faire pour *n* disques. En effet, si les n-1 disques sont déjà posés sur la tige B, le dernier disque encore posé sur A est le plus gros disque. Il suffit alors de le déplacer sur la tige C qui est vide, puis de déplacer les *n-1* disques de la tige B vers la tige C.



Déplacer les disques (autres que le plus gros) sur B en utilisant B comme disque transitoire.

HANOI A C B (n-1)
Déplacer le disque restant vers C

MOUVEMENT A C

Déplacer les disques (autres que le plus gros) vers C en utilisant A comme disque transitoire.

HANOI B A C (n-1)

```
#let mouvement de vers =
let s = "Deplace un disque de la tige " ^ de ^ "
vers " ^ vers in
begin
print_string(s);
print_newline()
end;;
mouvement : string -> string -> unit = <fun>
```

```
let rec hanoi depart milieu arrivee n =
if n=0 then () else
begin
hanoi depart arrivee milieu (n-1);
mouvement depart arrivee;
hanoi milieu depart arrivee (n-1);
end;;
#hanoi : string -> string -> string -> int -> unit =
<fun>
```

```
hanoi "a" "b" "c" 3;;

#Deplace un disque de la tige a vers c
Deplace un disque de la tige a vers b
Deplace un disque de la tige c vers b
Deplace un disque de la tige a vers c
Deplace un disque de la tige b vers a
Deplace un disque de la tige b vers c
Deplace un disque de la tige a vers c
- : unit = ()
```

Une autre version du programme « Hanoi » se trouve dans « *`[...]caml\examples\hanoi`* ». Il faut charger le programme « `loadall.ml` » puis lancer « `jeu` *x* » avec *x* = nombre de disques.

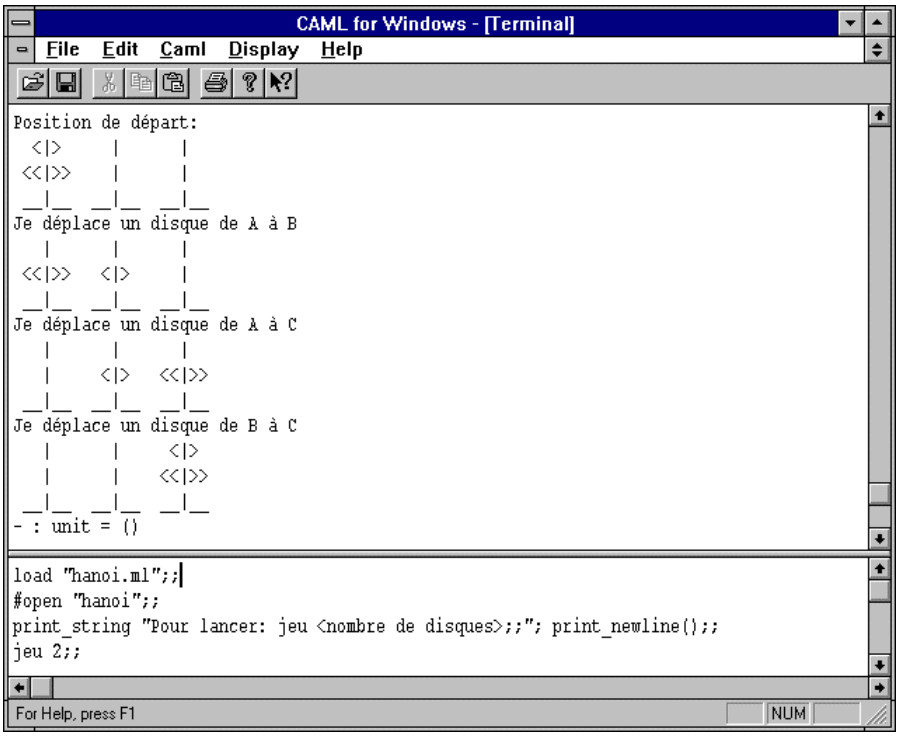


Figure 4 : Le programme Hanoï.

3.) Exercices.

EXERCICE 5.1

Construire en utilisant une fonctionnelle la fonction définie sur N* par :

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + ... + \frac{1}{n}$$

EXERCICE 5.2

On considère la suite définie par :

$U_1=0$, $U_2=1$ et $U_n=3*U_{n-1}+2*U_{n-2}$

Ecrire un programme Caml qui calcule la valeur et le rang du premier terme de cette suite supérieur ou égal à 10000 en utilisant une fonctionnelle.

EXERCICE 5.3

Ecrivez une fonction récursive qui multiplie deux nombres entiers (N1 et N2) en utilisant la méthode suivante :

On place des 2 nombres sur 2 colonnes puis on double le facteur de gauche tout en divisant par 2 celui de droite (quotient entier) jusqu'à obtenir 1 à droite. La somme des nombres qui sont placés à gauche d'un nombre impair donne alors le résultat.

Exemple :

Multiplication de 21 par 12 :

21	12
42	6
84	3
168	1
252	

EXERCICE 5.4

Etudiez le fonctionnement du tri "shuttle" (*shuttle* = navette) :

```
#let echange vecteur i j =
let a = vecteur.(i) in
begin
 vecteur.(i)<-vecteur.(j);
 vecteur.(j)<-a
end;;
echange : 'a vect -> int -> int -> unit = <fun>
#let tri_shuttle vecteur =
let test = ref true and j = ref 1 in
for i=0 to (vect_length(vecteur)-2) do
 j:=i;
 test:=(vecteur.(!j)>vecteur.(!j+1));
 while !test do
       echange vecteur !j (!j+1);
       j:=!j-1;
       if (!j<0) then test:=false
       else
       test:=vecteur.(!j)>vecteur.(!j+1);
 done;
done;;
tri_shuttle : 'a vect -> unit = <fun>
```

Pour cela, regardez l'évolution pas à pas du vecteur suivant :

5	3	6	8	6	1	9	7	2	4

Le vecteur t est un ensemble de dates (int*int*int) qui contient, par exemple, les éléments suivants :

```
#let t = [|(1854,10,20); (1768,09,04); (1844,03,30); (1802,02,26);
(1799,05,20); (1905,06,21);(1524,09,01)|];;
```

Evidemment, (1854,10,20) correspond au 20 octobre 1854. Transformez la fonction tri_shuttle en une fonctionnelle et écrivez la fonction à lui passer en paramètre de manière à ce qu'elle puisse trier le vecteur t.