«Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche
Le chef le mit en miettes
Les autres chiens en ce voyant
Vite, vite l'ensevelirent (...)
Au pied d'une croix en bois blanc
Où le passant pouvait lire :
Un chien vint dans l'office
Et prit une andouillette
Alors à coups de louche... »
Samuel Beckett, « En attendant Godot, Acte II ».

1.) Le filtrage.

En plus de la conditionnelle, Caml dispose d'une autre implémentation de l'alternative appelée l'analyse de cas. L'analyse de cas porte le nom technique de filtrage.

```
pattern  ::=  ident
           |  _
           |  pattern as ident
           |  ( pattern )
           |  ( pattern : typexpr )
           |  pattern | pattern
           |  constant
           |  ncconstr pattern
           |  pattern , pattern { , pattern}
           |  { label = pattern {; label = pattern } }

           |  [ ]
           |  [ pattern {; pattern} ]
           |  pattern :: pattern
```

`#let egal_un = function`
`| 1->true`
`| _-> false;;`
`egal_un : int->bool = <fun>`
	Le symbole '_' signifie "dans tous les autres cas".

`#let xor = function`
`| (false,false) -> false`
`| (false,true)  -> true`
`| (true,false)  -> true`
`| (true,true)   -> false;;`
`xor : bool * bool -> bool = <fun>`	

`#let xor = fun`
`| false false -> false`
`| false true  -> true`
`| true false  -> true`
`| true true   -> false;;`
`xor : bool -> bool -> bool = <fun>`
	On voit ici la différence entre *"fun"* et *"function"*. *"fun"* étant curryfiée, elle peut abstraire plusieurs arguments tandis que *"function"* ne peut abstraire qu'un seul argument.

`#let est_chiffre = function`
`| `0` | `1` | `2` | `3` | `4` | `5` | `6` | `7` | `8` | `9` -> true`
`| _ -> false;;`
`est_chiffre : char -> bool = <fun>`
	Le signe "|" représente le "ou" logique.

`#let est_chiffre = function`
`| `0` .. `9` -> true`
`| _ -> false;;`
`est_chiffre : char -> bool = <fun>`
	Dans les filtres, Caml Light reconnaît la forme ` a ` .. ` g ` (2 caractères constants séparés par ..) comme un raccourci du filtre
	
	` a ` | ` a1 ` | ` a2 ` |...| ` an ` | ` g `
	
	où a1,a2,...,an sont les caractères qui sont entre a et g dans l'ordre des caractères ASCII.

`#let est_pair = function`
`| 1 -> 0`
`| 2 -> 0`
`| _ as retour -> retour * 2;;`
`est_pair : int -> int = <fun>`
	Les alias permettent de lier un nom au contenu d'un filtre.

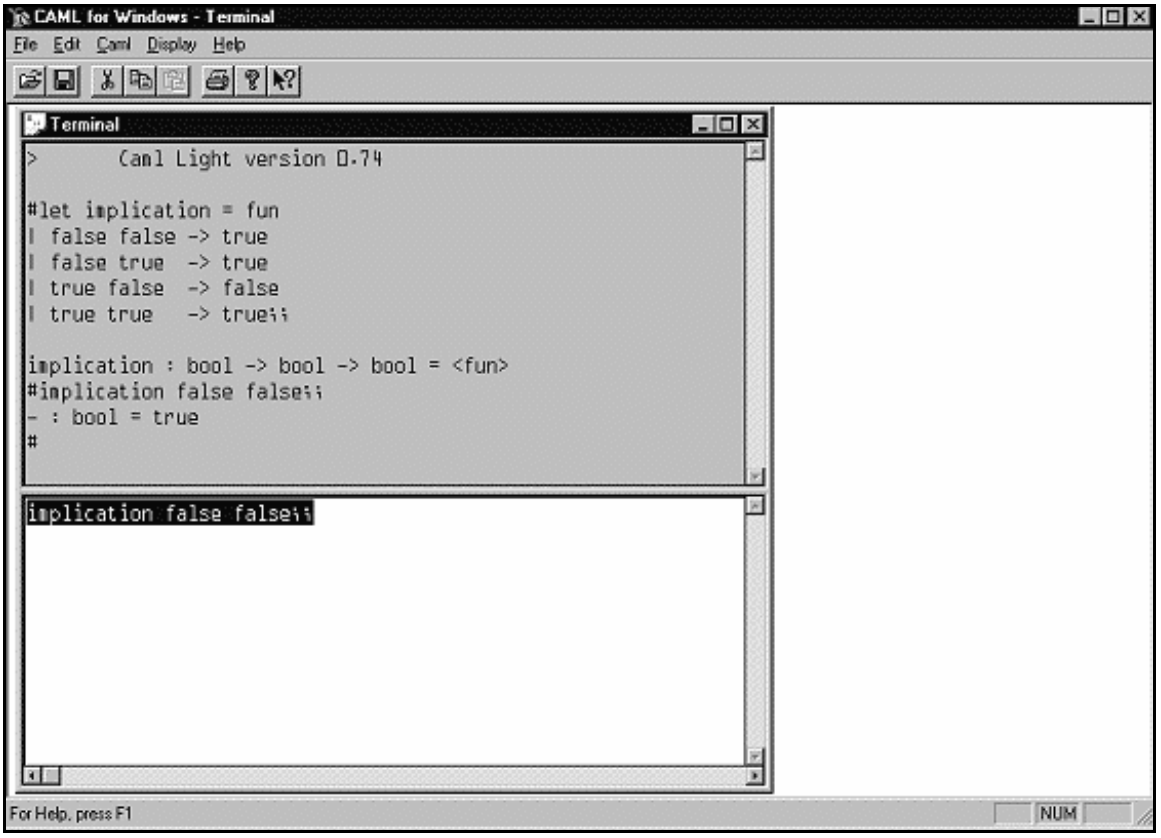


Figure 1 : Le filtrage en Caml.

1.1.) Les filtres gardés.

`#let pair = function`
`| x when x mod 2 = 1 -> false`
`| _ -> true;;`
`pair : int -> bool = <fun>`
	Les filtres gardés permettent d'inclure des conditions.

2.) La récursivité.

«J'ai épousé une veuve qui avait une fille adulte. Mon père, qui nous rendait souvent visite devint amoureux de ma (belle) fille et l'épousa. Dès lors, mon père devint mon gendre et ma (belle) fille devint ma (belle) mère. Quelques mois plus tard, ma femme donna naissance à un fils qui devint le beau-frère de mon père et à ce titre aussi mon oncle par alliance. La femme de mon père - qui est ma (belle) fille - a donné aussi naissance à un fils. Du coup, j'avais un (demi) frère et en même temps un petit-fils. Ma femme est ma grand-mère puisqu'elle est la mère de ma mère (par alliance). Donc je suis le mari de ma femme mais aussi son petit-fils (par alliance). En d'autres termes, je suis mon propre grand-père.»

N. Wirth, « Algorithm + data structures = programs », 1976.

2.1.) Notion de récursivité.

On appelle fonction récursive toute fonction qui, dans sa définition, fait appel à son propre identificateur. On peut ainsi définir la fonction somme par :

$$somme(n) = \begin{cases} 0 \quad si \quad n = 0 \\ \\ n + somme(n-1) \quad si \quad n > 0 \end{cases}$$

On pourrait dont écrire la fonction somme de la manière suivante (en pseudo-code) :

```
somme n = si n=0 alors 0 sinon n + somme (n-1)
```

On a donc :

```
Somme (4)  =   4 + somme(4 - 1) = 4 + somme(3)
           =   4+3+somme(3-1) = 4+3+somme(2)
           =   4+3+2+somme(2-1)=4+3+2+somme(1)
           =   4+3+2+1+somme(1-1) = 4+3+2+somme(1)
           =   4+3+2+1+0 = 10
```

De la même manière, on pourrait définir la multiplication de x par n ($n \in N^+$)de la manière suivante[1] :

$$multiplication(x,n) = \begin{cases} 0 \quad si \quad n = 0 \\ x + multiplication(x, n-1) \quad si \quad n > 0 \end{cases}$$

On aurait donc :

```
mult (4,3)  =   4 + mult(4,3-1) = 4 + mult(4,2)
            =   4 + 4 +mult(4,2-1) = 8 + mult(4,1)
            =   8 + 4 +mult(4,1-1) = 12 + mult(4,0)
            =   12 + 0 = 12
```

2.2.) La construction "let rec".

`let rec somme x =` `if x <= 0 then 0 else x + somme(x - 1);;`	`Calcul de la somme des nombres de 0 à x.`
`let rec mult x n =` `if n = 0 then 0 else x + mult x (n-1);;`	`Calcul du produit de x par n`

La fonction factorielle fonctionne selon le même principe :

$$fact(n) = \begin{cases} 1 \quad si \quad n <= 1 \\ n * fact(n-1) \quad si \quad n > 1 \end{cases}$$

`let rec fact n =` `if n<=1 then 1 else n * fact (n - 1);;`	Calcul de la factorielle de n

`#let rec fact = function`
`| 0->1`
`| n -> n*fact(n-1);;`
`fact : int->int = <fun>`
	La même chose en utilisant le filtrage.

[1] Pour n positif ou négatif ($n \in N$), il faudrait écrire :

$$multiplication(x,n) = \begin{cases} 0 \quad si \quad n = 0 \\ x + multiplication(x, n-1) \quad si \quad n > 0 \\ -x + multiplication(x, n+1) \quad si \quad n < 0 \end{cases}$$

2.3.) Terminaison.

On dira pour une fonction *f* et un argument *x* que *f(x)* termine si ce calcul nécessite un nombre fini d'opérations élémentaires.

Prouver la terminaison d'une fonction consiste à montrer que l'on arrive toujours au traitement des cas de base en un nombre finis d'appels. Pour cela, on peut utiliser la notion intuitive suivante :

une fonction récursive termine si on dispose d'une quantité liée à la fonction récursive et à ses arguments telle que :

- *cette quantité évolue dans un ensemble ordonné qui ne comporte pas de suite infinie strictement décroissante.*
- *à chaque appel de la fonction cette quantité décroît strictement.*
- *la fonction récursive achève son calcul pour les cas de base.*

2.4.) Déboguage.

Il n'existe pas, en Caml, de débogueur permettant d'exécuter un programme pas à pas. Par contre, il existe la fonction `trace` qui permet d'afficher tous les appels à une fonction donnée. L'argument de trace est le nom de la fonction en question. L'appel d'une fonction est indiqué par le signe `<--` suivi de l'argument concerné tandis que le signe `-->` signale un retour de fonction et affiche le résultat obtenu.

```
#trace "somme";;
La fonction somme est dorénavant tracée.
- : unit = ()
#somme 4;;
somme <-- 4
somme <-- 3
somme <-- 2
somme <-- 1
somme <-- 0
somme --> 0
somme --> 1
somme --> 3
somme --> 6
somme --> 10
- : int = 10
#trace "fact";;
La fonction fact est dorénavant tracée.
- : unit = ()
#fact 4;;
fact <-- 4
fact <-- 3
fact <-- 2
fact <-- 1
fact --> 1
fact --> 2
fact --> 6
fact --> 24
```

```
- : int = 24
```

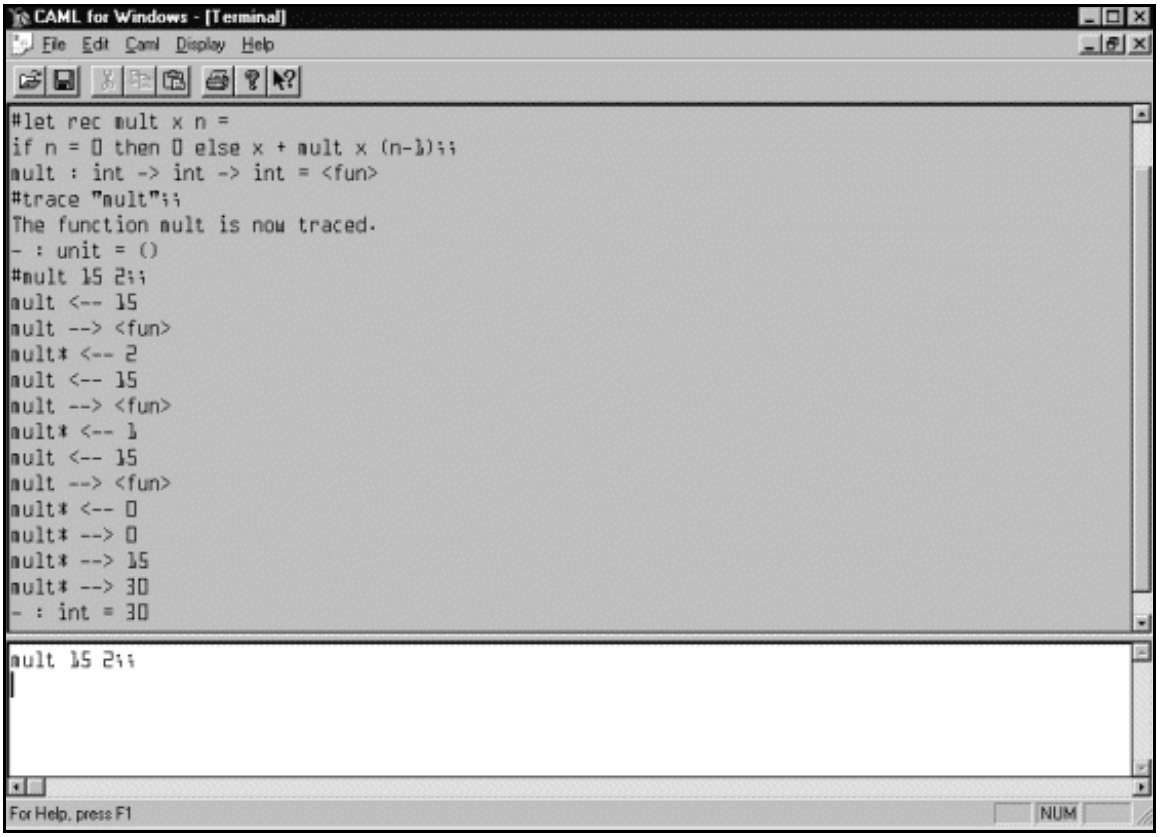


Figure 2 : Utilisation de la fonction "trace".

La fonction `untrace` permet d'annuler l'appel à trace.

```
#untrace "fact";;
La fonction fact n'est plus tracée.
- : unit = ()
```

2.5.) Récursivité croisée ou récursivité mutuelle.

Les fonctions mutuellement récursives sont définies simultanément en une seule instruction. Par exemple :

```
#let rec tic() = print_string "tic";tac();
and tac()=print_string "tac";tic();;
tic : unit -> 'a = <fun>
tac : unit -> 'a = <fun>
```

ou encore :

```
#let rec pair = function
| 0 -> true
| n -> impair(n-1)
and impair = function
| 0 -> false
| n -> pair(n-1);;
```

```
pair : int -> bool = <fun>
impair : int -> bool = <fun>
```

Dans l'algorithme de Salamin, le m-ième approximant de π s'obtient par la formule :

$$\pi_m = \frac{4a_m^2}{1-2\sum_{n=1}^{m} 2^n\left(a_n^2 - b_n^2\right)}$$

où a et b sont calculés par les récurrences :

$$a_{n+1} = \frac{1}{2}\left(a_n + b_n\right) \text{ et } b_{n+1} = \sqrt{a_n b_n} \text{ avec } a_0 = 1 \text{ et } b_0 = \frac{1}{\sqrt{2}}$$

On utilisera donc les fonctions mutuellement récursives a et b :

$$\begin{cases} a_0 = 1 \\ a_n = \dfrac{a_{n-1} + b_{n-1}}{2} \end{cases} \quad \text{et} \quad \begin{cases} b_0 = \dfrac{1}{\sqrt{2}} \\ b_n = \sqrt{a_{n-1} * b_{n-1}} \end{cases}$$

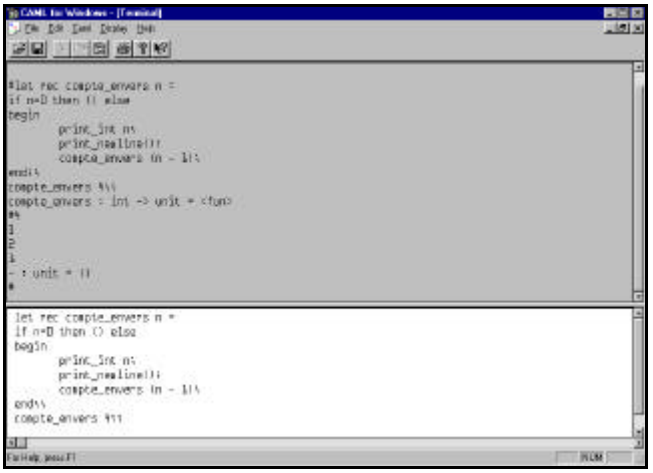
Ce qui donne en Caml :

```
#let rec a = function
| 0->1.
|n->(a (n-1) +. b(n-1)) /. 2.
and g = function
|0->1. /. sqrt(2.)
|n->sqrt(a (n-1) *. b(n - 1));;
a : int -> float = <fun>
b : int -> float = <fun>
```

2.6.) A l'endroit et à l'envers.

Soit la fonction « `compte_envers` »définie ainsi :

```
#let rec compte_envers n =
if n=0 then () else
begin
      print_int n;
      print_newline();
      compte_envers (n - 1);
end;;
compte_envers : int -> unit = <fun>
```

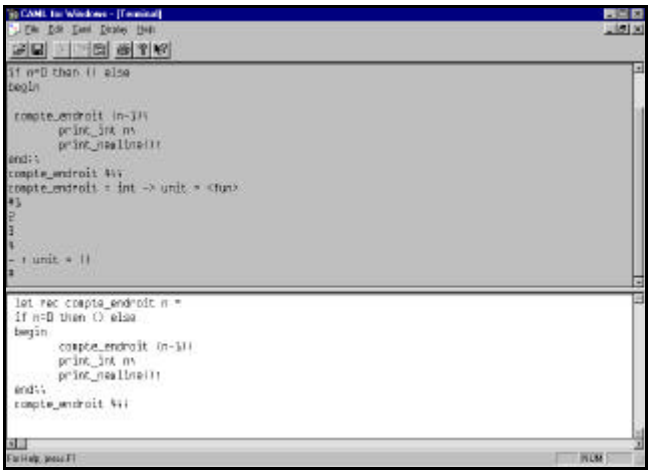


On remarque qu'à l'exécution de la fonction, l'impression se produit **avant** l'appel récursif. La fonction affichera donc successivement 4, 3, 2 puis 1.

Par contre, si on modifie la fonction en plaçant l'impression après l'appel récursif :

```
#let rec compte_endroit n =
if n=0 then () else
begin

 compte_endroit (n-1);
      print_int n;
      print_newline();
end;;
compte_endroit : int -> unit = <fun>
```



On remarque que cette fois-ci, l'impression se produit **après** l'appel récursif. La fonction affichera donc successivement 1, 2, 3 puis 4.

3.) Types de données abstrait.

Un **type de données** est une collection d'objets qui ont des propriétés identiques indépendamment de toute représentation.

Un type de données abstrait est un type de données muni d'une signature (c'est à dire de fonctions de manipulation typées) et accompagné de sa sémantique (qui permet de décrire précisément son fonctionnement).

4.) Les piles.

La pile est un type de données abstrait particulier dans lequel toute insertion ou suppression d'un élément se fait à une extrémité appelée *dessus* ou *sommet de la pile*. L'appellation anglo-saxonne consacrée pour les piles est "*LIFO list*" ou liste "*dernier entré premier sorti*". Ceci s'explique intuitivement en comparant une pile à une pile d'assiettes sur une étagère : si on se refuse à manipuler plus d'un élément à la fois, les seules actions possibles sont l'ajout (empilement) ou le retrait (dépilement) d'un élément.

Une application importante des piles est la mise en œuvre de la récursivité dans les applications. Tous les langages comme Caml ou Pascal qui permettent l'emploi de la récursivité utilisent une pile pour garder une trace de l'état de chaque variable de chaque procédure rendue active à un moment donné. Ces états portent le nom de contexte. Quand une procédure P est appelée, son contexte est placé au sommet de la pile. Si P appelle procédure P1, le contexte de P1 sera placé au sommet de la pile. A la fin de l'exécution de P, son contexte devra se trouver au sommet de la pile puisque ont ne peut revenir à P avant que toutes les autres procédures ne lui aient "rendu la main". Voici un exemple en appelant la fonction somme(5) :

```
#let rec somme n =
if n = 1 then
begin
      print_string "renvoyer 1";
      print_newline();
      1;
end
else
begin
      print_string "empiler ";
      print_int n;
      print_newline();
      let a = n * somme(n-1) in
      begin
      print_string "depiler ";
      print_int n;
      print_string " renvoyer ";
      print_int a;
      print_newline();
      a;
      end;
end;;
somme : int -> int = <fun>

somme 5;;
empiler 5
empiler 4
empiler 3
empiler 2
renvoyer 1
depiler 2 renvoyer 2
depiler 3 renvoyer 6
depiler 4 renvoyer 24
depiler 5 renvoyer 120
- : int = 120
```

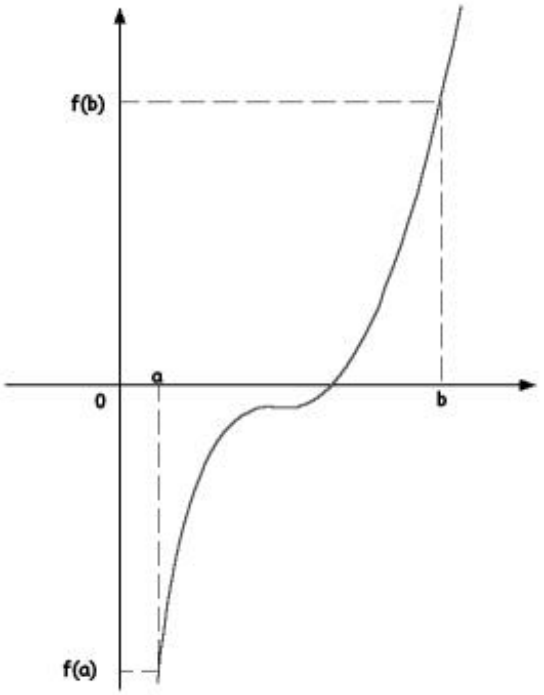
5.) Résolution d'équations.

On se propose de résoudre l'équation $f(x)=0$ dans laquelle f est une fonction numérique à variable réelle.

5.1.) Méthode de la dichotomie.

Pour celle méthode, on suppose que la racine x de l'équation $f(x)=0$ est comprise dans l'intervalle $[a,b]$ et que f est continue dans l'intervalle $[a,b]$.

La méthode de la dichotomie se fonde sur le théorème des « *valeurs intermédiaires* ». Ce théorème affirme que si une fonction f est continue sur l'intervalle $[a,b]$ et que si les valeurs de f*(a)* et *f(b)* sont l'une positive et l'autre négative, alors il existe au moins une valeur x dans $[a,b]$ pour laquelle *f(x)* = 0.



Cette méthode consiste à diviser successivement l'intervalle de recherche $[a,b]$ en ramenant sa borne inférieure ou sa borne supérieure à la valeur moyenne de l'intervalle, selon que la racine de l'équation est comprise dans la moitié supérieure ou inférieure, respectivement, de l'intervalle. On a donc une suite de valeurs x_i vérifiant

$$x_0 = \frac{a+b}{2},\quad x_1 = \frac{a_1+b_1}{2},\quad \ldots x_n = \frac{a_n+b_n}{2}$$

Le calcul des a_i et b_i s'effectue selon :

$$\alpha \in [a_{i-1}, x_{i-1}] \Rightarrow a_i = a_{i-1},\ b_i = x_{i-1}$$
$$\alpha \in [x_{i-1}, b_{i-1}] \Rightarrow a_i = x_{i-1},\ b_i = b_{i-1}$$

avec $a_0 = a$ et $b_0 = b$.

La première condition est réalisée lorsque la fonction f change de signe dans l'intervalle $[a_{i-1}, x_{i-1}]$; la seconde lorsque f change de signe dans l'intervalle $[x_{i-1}, b_{i-1}]$. Le nombre d'itérations à réaliser pour approcher la racine de l'équation donnée est déterminé par l'inégalité :

$$|\alpha - x_n| < \frac{(b-a)}{2^{n+1}}$$

On a donc la fonction dichotomie qui retourne, si elle existe, la racine de l'équation passée en argument dans l'intervalle $[a,b]$ et avec la précision Epsilon passée en arguments.

Par exemple, si on recherche la racine carrée d'un nombre k (c'est à dire si on cherche à résoudre l'équation « $x^2 - k = 0$ ») on écrira :

```
#let carre nombre = nombre *. nombre;;
carre : float -> float = <fun>
#let rec dicho borne_inf borne_sup precision k =
 let equation n = (carre n ) -. 2. and racine = (borne_inf +.
borne_sup) /. 2. in
  if (borne_sup -. borne_inf) > precision then
    if equation(borne_inf)*. equation(racine) < 0. then
      dicho borne_inf racine precision k
    else
      dicho racine borne_sup precision k
  else racine;;
dicho : float -> float -> float -> 'a -> float = <fun>
```

Donc, si on recherche, dans l'intervalle [0.,2.] et avec une précision de 10^{-10} la racine de 2, on appliquera la fonction `dicho` ainsi :

```
#dicho 0. 2. 1e-10 2.;;
- : float = 1.41421356236
```

5.2.) Méthode de la sécante.

La méthode de la sécante[2] consiste à tracer la droite AB passant par les points $(a, f(a))$ et $(b, f(b))$. Cette droite coupe l'axe des x en un point x_0 tel que :

$$x_0 = \frac{bf(a) - af(b)}{f(a) - f(b)}$$

Si la droite AB est au-dessus de la courbe de f, c'est à dire si $f(x_0) < 0$, on trace la droite passant par $(x_0, f(x_0))$ et $(b, f(b))$. On obtient un nouveau point d'intersection x_1 avec l'axe des x et on répète indéfiniment l'opération. Les x_n vérifient la relation :

$$x_{n+1} = \frac{bf(x_n) - x_n f(b)}{f(x_n) - f(b)}$$

Il est facile de voir que dans le cas où la corde AB est au-dessous de la courbe de f, c'est à dire si $f(x_0) > 0$, les x_n vérifient la relation suivante :

$$x_{n+1} = \frac{af(x_n) - x_n f(a)}{f(x_n) - f(a)}$$

On voit que cette méthode consiste à supposer que dans l'intervalle $[a,b]$ la fonction f est représentée approximativement par une droite.

[2] Aussi appelée « interpolation linéaire » ou « méthode des parties proportionnelles ».

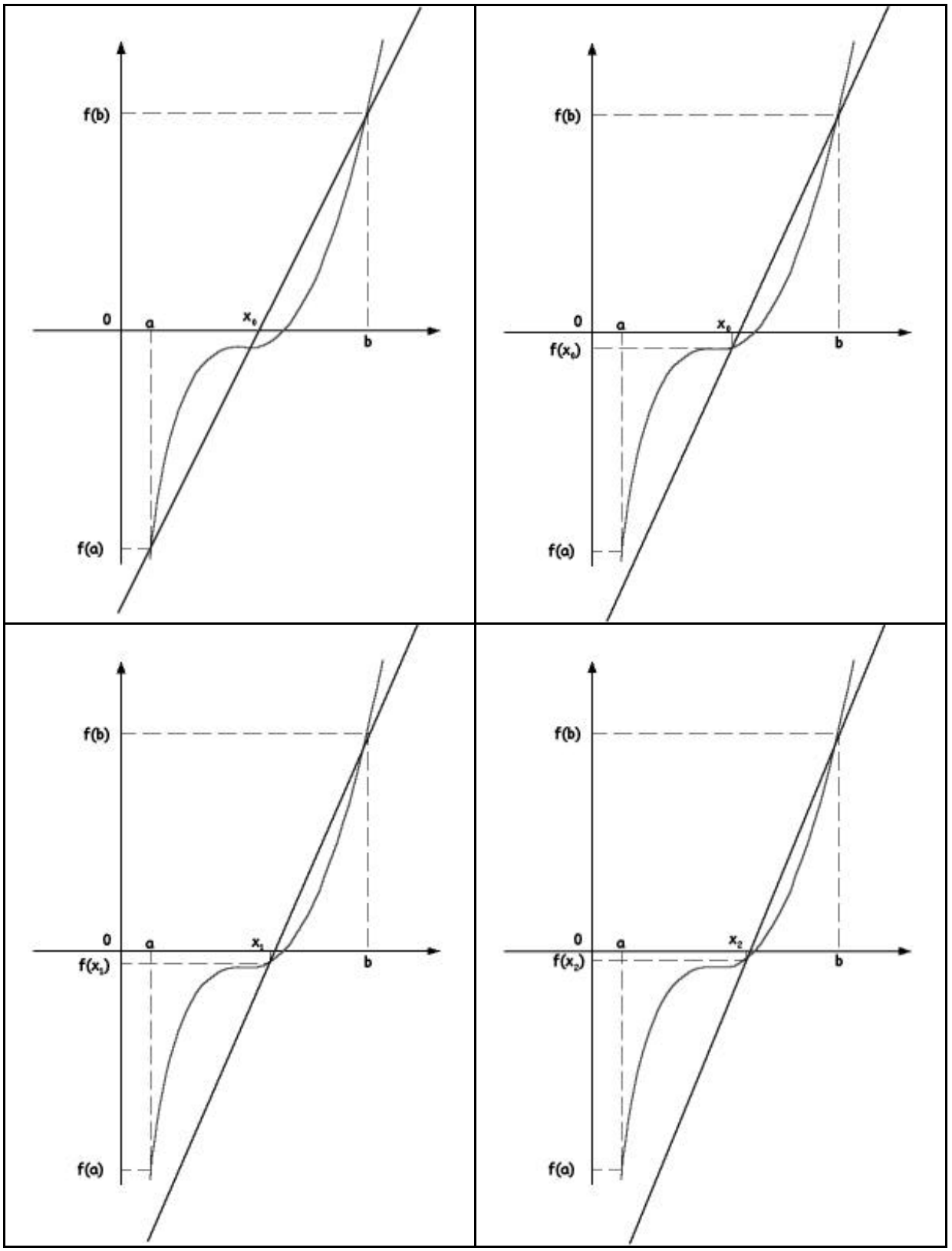


Figure 3 : Recherche d'une racine par la méthode de la sécante.

On démontre que lorsque n tend vers l'infini, x_n tend vers la racine de l'équation $f(x)=0$.

Par exemple, si on recherche la racine carrée d'un nombre k (c'est à dire si on cherche à résoudre

l'équation « $x^2 - k = 0$ ») on écrira :

```
#let rec secante min max k nb =
if nb=1 then min else
let f x = x *. x  -. k  in
let fa = f (min) and fb = f (max) in
let x0 = ((max *. fa) -. (min *. fb)) /. (fa -. fb) in
let fx0 = f x0 in
if fx0 < 0. then secante x0 max k (nb - 1) else
secante min x0 k (nb - 1);;
secante : float -> float -> float -> int -> float = <fun>
```

Par exemple, pour calculer la racine de 2 (qu'on estime comprise entre 0 et 2), en 50 itérations, on écrira :

```
#secante 0. 2. 2. 50;;
- : float = 1.414213562372
```

5.3.) L'algorithme de Newton/Raphson.

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

$$[a\ ,\ b\ ]$$

par la récurrence

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

L'application de cette méthode à la fonction

$$f(x) = x^2 - a$$

conduit à la suite récurrente :

$$y_{n+1} = y_n - \frac{{y_n}^2 - a}{2y_n}$$

qui permet le calcul de $\sqrt{a}$.

```
#let rec racine a n =
if n=1 then 1. else
let x = (racine a (n-1)) in
x -. (((x *. x) -. a) /. (2. *. x));;
racine : float -> int -> float = <fun>
```

Par exemple, pour calculer la racine de 2 en 50 itérations, on écrira :

```
#racine 2. 50;;
- : float = 1.41421356237
```

6.) Exercices.

EXERCICE A:

Calculer le $n_{ième}$ nombre de Lucas défini par :

$L_1 = 1,\ L_2 = 3$ et $L_n = L_{n-1}+L_{n-2}$

```
let rec lucas = function n->
if n=1 then 1 else
if n=2 then 3 else lucas(n-1) + lucas(n-2);;
```

EXERCICE B :

En utilisant l'algorithme de Newton / Raphson, écrire une fonction calculant la racine cubique d'un réel.

$f(x) = x^3 - a = 0$
$f'(x) = 2x^2$
$y_{n+1} = y_n - (y_n^3 - a) / (2y_n^2).$

```
#let rec cubique a n =
if n<=1 then 1. else
let x = (cubique a (n-1)) in
x -. (((x *. x *. x) -. a) /. (2. *. (x *. x)));;
cubique : float -> int -> float = <fun>
```

EXERCICE C :

La division euclidienne est une division particulière qui s'applique aux entiers relatifs de l'ensemble $\mathbb{Z}$. Elle se définit de la manière suivante : si a et b sont deux entiers relatifs, b étant non nul, il existe deux entiers relatifs uniques q et r tels que $a = bq + r$, avec $0 \le r \le b - 1$. q est appelé le quotient de la division de a par b, et r le reste.

Ecrivez en Caml une fonction récursive qui reçoit en paramètres deux entiers a et b et retourne la paire (q,r) où q est le quotient entier de a par b et r le reste. Bien entendu, cette fonction ne devra utiliser que l'addition et la soustraction.

La division euclidienne peut s'exprimer ainsi :

$$euclide(a,b) = \begin{cases} (0,b) \;\; si \;\; b > a \\ (1, a-b) \;\; si \;\; b > (a-b) \\ (1,0) + euclide(a-b, b) \quad \text{sin } on. \end{cases}$$

Ce qui se traduit facilement en Caml :

```
#let add (a,b) (c,d) = (a+c,b+d);;
add : int * int -> int * int -> int * int = <fun>
#let rec euclide = fun
| a b when b>a -> (0,b)
| a b when b > (a-b) -> (1,a-b)
| a b -> add (1,0) euclide (a-b) b;;
```

EXERCICE D :

La constante de Vijayaraghavan est la racine de l'équation $x^3=x+1$. La déterminer par la méthode de Newton.

On cherche à résoudre l'équation $f(x) = x^3-x-1 = 0$. $f'(x) = 2x^2-1$ par la récurrence :

$$xn+1 = xn - (xn3 - x - 1) / (2xn^2 -1).$$

```
#let rec racine n =
if n=1 then 1. else
let x = (racine (n-1)) in
x -. (((x *. x *. x) -. x -. 1.) /. (2. *. x *. x -. 1.));;
racine : int -> float = <fun>
#racine 501;;
- : float = 1.32471795724
```

EXERCICE E :

L'empereur Frédéric II posa à Léonard de Pise le problème suivant (cette sorte de compétition entre scientifiques devait se développer au XVIe et au XVIIe siècle) : résoudre l'équation du troisième degré $x^3 + 2x^2 + 10x = 20$. Trouver la solution par la méthode de Newton.

On cherche à résoudre l'équation $f(x) = x^3+2x^2+10x-20 = 0$. $f'(x) = 2x^2+2x+10$ par la récurrence :

$$xn+1 = xn - (x^3+2x^2+10x-20) / (2x^2+2x+10).$$

```
#let rec racine n =
if n=1 then 1. else
let x = (racine (n-1)) in
x -. (((x *. x *. x) +. 2. *. (x *. x) +. 10. *. x -. 20.) /. (2. *.
x *. x +. 2. *. x +. 10.));;
racine : int -> float = <fun>
#let s = racine 500;;
s : float = 1.36880810782
```

EXERCICE F :

La clé "C" d'un numéro Insee N est égale à (97-(N MODULO 97)). Ecrivez une fonction récursive qui accepte en entrée une chaîne de chiffres et calcule "C".

```
#let rec cle = fun
| s r when string_length(s)=0 -> 97 - r
| s r -> let i = r*10 + int_of_string (sub_string s 0 1 ) in  cle
(sub_string s 1 ((string_length s)-1)) (i mod 97);;
cle : string -> int -> int = <fun>
```

EXERCICE G :

On considère la fonction u suivante :

```
#let rec u n =
if n = 0 then 0 else n - u(n-1);;
u : int -> int = <fun>
```

La fonction *u* termine-t-elle ?

EXERCICE H :

Ecrire une fonction qui teste si un nombre est premier, en utilisant le théorème de Wilson : *n* est premier si et seulement si *n* divise exactement *(n-1)! + 1*, c'est à dire si $\frac{(n-1)!+1}{n}$ est un entier.

```
#let rec fact = function
| 0 -> 1
| n -> n * fact (n-1);;
fact : int -> int = <fun>
#let Wilson n =
let tmp = (fact (n - 1)) + 1 in
let reste = tmp mod n in
if reste=0 then true else false;;
Wilson : int -> bool = <fun>
```

EXERCICE I :

Une règle récursive exprime y_n à l'aide d'un ou de plusieurs de ces prédécesseurs. Par exemple, si y_n est le nombre de façons d'asseoir au moins une personne sur une rangée de *n* sièges de telle sorte que deux personnes n'occupent jamais deux chaises voisines, on peut démontrer que $y_n = y_{n-1} + y_{n-2} + 1$ avec $y_1 = 1$ et $y_2 = 2$. Ecrire une fonction récursive qui calcule y_n en fonction de n.

```
#let rec nb_personne = function
| 1 -> 1
| 2 -> 2
| x -> nb_personne (x-1) + nb_personne (x-2) +1;;
nb_personne : int -> int = <fun>
```

6.1.) Exercices.

EXERCICE 3.1.

Léonard de Pise, également appelé Léonardo Fibonacci, présenta en 1202 le problème suivant : "*Un homme a placé un couple de lapins dans un enclos. Combien de couples de lapins peuvent être produits par celui-ci en un an si on suppose que tous les mois chaque couple engendre un nouveau couple lequel devient alors productif à partir du second mois* ?". Le nombre de lapins progresse selon une suite récurrente dite "de Fibonacci" définie par :

$U_1 = 1,\ U_2 = 2$ et $U_n = U_{n-1} + U_{n-2}$

Calculez le $n^{\text{ième}}$ terme de la suite de Fibonacci.

EXERCICE 3.2.

Calculer le PGCD de deux nombres entier a et b en utilisant l'algorithme d'Euclide :

$$p\gcd(a,b) = \begin{cases} a & si \quad a = b \\ p\gcd(a-b,b) & si \quad a > b \\ p\gcd(a,b-a) & si \quad a < b \end{cases}$$

EXERCICE 3.3

Construire récursivement en Caml la fonction définie sur N* par :

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

EXERCICE 3.4

Définir la fonction récursive "puissance" qui reçoit en argument x et n (n>=0) et calcule x^n. A l'aide de la directive #infix[3], rendez cette fonction utilisable sous la forme "x puissance n".

EXERCICE 3.5

Réaliser une fonction récursive donnant la chaîne de chiffres binaires correspondants à un entier. On utilisera le principe suivant : pour transcrire un nombre n exprimé en base 10 en un nombre en base b, il suffit de diviser n par b, puis de diviser ce quotient par b, puis le nouveau quotient obtenu par b, et ainsi de suite jusqu'à obtention du quotient 0. Les restes successifs sont les chiffres de n exprimés en base b. Par exemple, pour exprimer 249 (base 10) en base 2, on écrit :

249/2	=	124	reste	**1**
124/2	=	62	reste	**0**
62/2	=	31	reste	**0**
31/2	=	15	reste	**1**
15/2	=	7	reste	**1**
7/2	=	3	reste	**1**
3/2	=	1	reste	**1**
1/2	=	0	reste	**1**

Le nombre $249_{(\text{base } 10)}$ s'écrira $11111001_{(\text{base } 2)}$.

[3]La directive `#infix " id "` change le statut de l'identifiant `id` de manière à ce qu'il soit reconnu comme un opérateur infixé, c'est à dire un opérateur se trouvant entre ses paramètres, comme c'est le cas de `+ , /, *`…

Exemple :

```
#let add_mod_10 a b = (a+b) mod 10;;
add_mod_10 : int -> int -> int = <fun>
#add_mod_10 130 121;;
- : int = 1
##infix "add_mod_10";;
#130 add_mod_10 121;;
- : int = 1
```