

# AIDE-MÉMOIRE CAML

1	GÉNÉRALITÉS . . . . .	3
1.1	Présentation rapide de Caml . . . . .	3
1.2	La bibliothèque de modules . . . . .	3
1.3	Chargement de morceaux de programmes avec <b>include</b> . . . . .	4
1.4	Identificateurs et mots réservés . . . . .	4
1.5	Déclarations avec <b>let</b> . . . . .	4
1.5.1	Déclarations globales . . . . .	4
1.5.2	Déclarations simultanées . . . . .	5
1.5.3	Déclarations locales d'identificateurs à l'intérieur d'expressions . . . . .	5
1.6	Séquences d'expressions . . . . .	5
2	FONCTIONS . . . . .	6
2.1	Fonction à un seul argument . . . . .	6
2.2	Fonctions de plusieurs variables . . . . .	6
2.2.1	Fonction définie sur un produit cartésien . . . . .	6
2.2.2	Déclaration curryfiée . . . . .	6
2.2.3	Gestion des parenthèses . . . . .	7
2.3	Fonctions utilisant le filtrage (ou pattern-matching) . . . . .	7
2.4	Fonctions récursives . . . . .	9
2.5	Conversion entre opérateurs binaires infixes et préfixes . . . . .	9
3	LES TYPES DE DONNÉES PRÉDÉFINIS . . . . .	10
3.1	Les types élémentaires . . . . .	10
3.1.1	Le type <b>unit</b> . . . . .	10
3.1.2	Les nombres entiers ( <b>int</b> ) . . . . .	10
3.1.3	Les nombres flottants ( <b>float</b> ) . . . . .	10
3.1.4	Les caractères ( <b>char</b> ) . . . . .	11
3.1.5	Les booléens ( <b>bool</b> ) . . . . .	11
3.2	Les types composés . . . . .	12
3.2.1	Les types fonctionnels . . . . .	12
3.2.2	Les produits cartésiens . . . . .	12
3.2.3	Les listes . . . . .	12
3.3	Types définissables par l'utilisateur . . . . .	14
3.3.1	Les types somme . . . . .	14
3.3.2	Les types produit (ou enregistrements) . . . . .	15
3.3.3	Types polymorphes (ou paramétrés) . . . . .	16
3.3.4	Renommer des types . . . . .	16
3.3.5	Imposer le type des fonctions . . . . .	17
3.4	Les types mutables . . . . .	17
3.4.1	Les références . . . . .	17
3.4.2	Les tableaux (ou vecteurs) . . . . .	18
3.4.3	Les chaînes de caractères ( <b>string</b> ) . . . . .	19
3.4.4	Les enregistrements à champs variables . . . . .	19
4	LES STRUCTURES DE CONTRÔLE . . . . .	21
4.1	L'expression conditionnelle <b>if ...then ...else</b> . . . . .	21
4.2	Les boucles . . . . .	21
4.2.1	La boucle inconditionnelle <b>for</b> . . . . .	21
4.2.2	La boucle conditionnelle <b>while</b> . . . . .	21
4.3	Les gardes . . . . .	21
5	LE GRAPHISME AVEC CAML . . . . .	23

5.1	Généralités . . . . .	23
5.2	Couleurs . . . . .	23
5.3	Tracés de points et de lignes . . . . .	24
5.4	L'exemple complet de la cardioïde . . . . .	24
5.5	Remplissage de figures . . . . .	24
5.6	Images . . . . .	25
5.7	Tracé de texte . . . . .	25
5.8	Sonder la souris et le clavier . . . . .	25
5.9	Émettre un son . . . . .	26
5.10	Sauver l'écran graphique . . . . .	26
6	DIVERS OUTILS CAML . . . . .	27
6.1	Mesurer le temps en Caml . . . . .	27
6.2	Le mode trace . . . . .	27
6.3	Les exceptions . . . . .	27
6.4	Générateur de nombres pseudo-aléatoires . . . . .	27
6.5	Les entrées & sorties . . . . .	27
6.5.1	Notion de canal . . . . .	27
6.5.2	Ouverture et fermeture de canaux . . . . .	28
6.5.3	Fonctions générales d'écriture . . . . .	28
6.5.4	Fonctions générales de lecture . . . . .	28
6.5.5	Fonctions d'affichage à l'écran . . . . .	29
6.5.6	Fonctions de lecture du clavier . . . . .	29
6.5.7	Un exemple complet . . . . .	29
6.6	Faire des calculs en précision arbitraire avec Caml . . . . .	30
6.7	Conseils de présentation des programmes . . . . .	31
7	AIDE-MÉMOIRE RÉSUMÉ D'EMACS . . . . .	33
7.1	L'indispensable . . . . .	33
7.2	Mieux comprendre le terminal . . . . .	33
7.3	Aller plus loin avec Emacs et Tuareg . . . . .	34
8	AIDE-MÉMOIRE RÉSUMÉ DE CAML . . . . .	35

## Auteurs

Guillaume HABERER, Philippe SAUNOIS, Daniel GENOUD – Lycée La MARTINIÈRE MONPLAISIR (LYON)

## À propos de ce document

Ce polycopié, qu'il faut apporter lors des cours et des travaux pratiques, rappelle la syntaxe des instructions les plus utilisées du langage (d'après le *Manuel de référence du langage Caml*, X. LEROY et P. WEIS). Plusieurs liens hypertextes sont disponibles sur la version électronique, en particulier pour des compléments présentés dans la documentation en ligne en anglais ou la faq en ligne en français.

# 1 GÉNÉRALITÉS

## 1.1 Présentation rapide de Caml

Caml est un acronyme signifiant : « Categorical Abstract Machine Language ».

Caml-light est distribué par l'INRIA, et disponible librement et gratuitement pour la plupart des plateformes.



Nous utilisons Caml-light en **mode interactif**, c'est-à-dire comme Maple, à la manière d'une calculette, dans une boucle infinie qui ne peut se terminer qu'avec l'instruction `quit();;` ou par la combinaison de touches `ctrl-d`

Chaque boucle comprend trois phases :

- *entrée* de la phrase Caml, puis lecture lorsqu'elle se termine par `;;` suivi de *Enter*
- *évaluation* (typage de la phrase, compilation, puis exécution)
- *réponse*, qui est en général l'affichage d'un résultat (ou d'un message d'erreur)

De façon plus précise, la forme de la réponse de Caml dépend de la phrase entrée, selon qu'il s'agit :

- d'une **déclaration** (ou *définition*) (précédée du mot-clé `let`)
- d'une **expression** dont l'évaluation produit un résultat autre que `()` ou produisant un *effet de bord* (voir ci-dessous)
- d'une **directive de compilation** précédée du symbole `#` (comme `#open`, `#close`, `#infix`, ...)

```
let a = 2;;           1 + 2;;           print_int 13;;           #open "random";;
(* a : int = 2 *)     (* - : int = 3 *)     (* 13 - : unit = () *)
```

Le symbole `-` qui apparaît parfois a le sens suivant : « l'expression précédente vaut » et Caml donne alors une réponse double de la forme `: type = valeur`.

Le mode interactif de Caml est surtout utile pour la mise au point de programmes devant être compilés plus tard : les réponses de typage ont surtout un intérêt pédagogique.

Une phrase Caml peut s'étendre sur plusieurs lignes. C'est toujours le terminateur `;;` qui signale à Caml où se termine cette phrase.

On appelle **effet de bord** une *expression* qui a pour valeur `()` (*rien*), dont l'action est de modifier l'environnement. Ainsi :

- écrire sur l'écran ou dans un fichier sur un disque,
- modifier des cases de la mémoire,
- dessiner sur l'écran graphique,
- émettre un son, etc.

Caml est un langage dans lequel chaque objet possède un type (unique).

Le typage est automatique, sans intervention du programmeur (sauf de rares exceptions).

La plupart des fonctions Caml opèrent sur des arguments et renvoient un résultat ayant chacun un type imposé d'avance (qui peut parfois être le type générique `'a`, c-à-d quelconque), ce qui permet de détecter les erreurs de syntaxe éventuelles durant la phase de compilation (donc avant l'exécution) grâce à un mécanisme sophistiqué de vérification des types.

Une description détaillée des types de Caml sera donnée dans la section 3.

Dans l'immédiat, citons les types de base suivants, utiles pour comprendre la section 2 relative aux fonctions :

`unit` (*rien*), `int` (*entier*), `float` (*flottant*), `bool` (*booléen*), `char` (*caractère*), `string` (*chaîne de caractères*).

## 1.2 La bibliothèque de modules

Le principe d'appel de définitions placées dans un module en Caml est le même que celui des *packages* de Maple : seule la syntaxe change. Avec Maple, si on veut utiliser par exemple la fonction `det`, soit on l'appelle avec le nom complet `linalg[det]`, soit on charge en mémoire tout le *package* `linalg` avec `with(linalg)`; auquel cas on a alors accès directement à la fonction `det`.

Supposons que le module `machin` contienne la définition (locale) de l'identificateur `ma_def`.

Alors le nom complet (ou global) de cet identificateur pour l'environnement de travail est `machin__ma_def` (nom du module, suivi de *deux symboles de soulignement*, puis du nom local de l'identificateur).

Pour utiliser celui-ci dans l'environnement, trois cas se présentent :

1. si le module `machin` fait partie des modules déjà ouverts, alors le nom `ma_def` suffit.
2. si le module `machin` n'est pas ouvert :
  - si on ne souhaite pas l'ouvrir, on l'appellera alors par son nom global complet `machin__ma_def`.

## 1. GÉNÉRALITÉS

↪ sinon on ouvre le module en entier grâce à la directive de compilation

`#open "machin";;` (noter les guillemets), ce qui ramène au 1<sup>er</sup> cas.

Plus d'informations sur les modules chargés par défaut et les modules à charger en cas de besoin au paragraphe « The Caml Light library » de la documentation en ligne

### 1.3 Chargement de morceaux de programmes avec include

Supposons avoir créé un morceau de programme Caml avec un éditeur de texte quelconque et l'avoir sauvé sur disque sous le nom `mon_programme.ml`. Il est alors possible de le faire exécuter dans l'environnement de travail comme si on venait de l'entrer au clavier grâce à l'appel de la fonction `include "mon_programme.ml";;` s'il se trouve dans le répertoire courant. Sinon, on fait précéder le nom par le chemin :

Linux ou MacOS X : `include "/donnees/repertoire_local/mon_programme.ml"`

Windows : `include "c:\\caml\\travail\\mon_programme.ml"`

Les phrases du fichier inclus ne sont pas reprises dans la fenêtre `*caml-toplevel*` : seuls les résultats des évaluations sont affichés.

**Remarque.** Toutes les déclarations effectuées dans le fichier inclus sont utilisables avec leur nom local.

### 1.4 Identificateurs et mots réservés

Les **commentaires** en Caml sont encadrés entre `(*` et `*)`.

On appelle « **blanc** » un des caractères suivants : espace, tabulation, retour chariot, saut de ligne.

Les **identificateurs** en Caml, qui sont des **noms** qu'on donne à des expressions, obéissent aux règles suivantes :

- ↪ ce sont des chaînes de caractères ne contenant pas de « blancs ».
- ↪ le premier caractère doit être une lettre (non accentuée) prise dans  $\{a, \dots, z, A, \dots, Z\}$
- ↪ les caractères suivants peuvent être des lettres, des lettres accentuées, des chiffres, le symbole `_` de soulignement ou l'accent aigu `'`, mais en revanche `+`, `&`, `>`, `=`, `%`, `...` sont interdits.

Caml fait la distinction entre majuscules et minuscules.

Les « blancs » servent à séparer des identificateurs ou des arguments de fonctions (en l'absence de parenthèses).

Une phrase Caml peut s'étendre sur plusieurs lignes : c'est le terminateur `;;` qui signale où se situe la fin de la phrase.

En outre, les mots réservés du langage ne peuvent pas être utilisés comme identificateurs. Il s'agit de :

<code>and</code>	<code>as</code>	<code>begin</code>	<code>do</code>	<code>done</code>	<code>downto</code>	<code>else</code>	<code>end</code>	<code>exception</code>	<code>for</code>	
<code>fun</code>	<code>function</code>	<code>if</code>	<code>in</code>	<code>let</code>	<code>match</code>	<code>mutable</code>	<code>not</code>	<code>of</code>	<code>or</code>	<code>prefix</code>
<code>rec</code>	<code>ref</code>	<code>then</code>	<code>to</code>	<code>try</code>	<code>type</code>	<code>value</code>	<code>when</code>	<code>where</code>	<code>while</code>	<code>with</code>

L'interpréteur distingue majuscule et minuscule.

Caml *calcule uniquement avec des valeurs* et non avec des quantités formelles comme le fait Maple.

Avec Caml, on ne peut pas appeler un nom s'il n'a pas été défini (c-à-d assigné) auparavant.

### 1.5 Déclarations avec let

#### 1.5.1 Déclarations globales

**Syntaxe** `let ident = valeur ;;`

Il s'agit d'une **liaison** d'un nom à une valeur (en quelque sorte une abréviation) analogue à celle pratiquée en Mathématiques lorsqu'on dit : « Posons *ident = valeur* »

En Caml, une définition (ou déclaration) *n'est pas modifiable* et est *en lecture seulement* : ici, *ident* n'est donc pas une « variable » au sens usuel du mot en informatique.

En résumé, le mécanisme du `let` est donc très différent de celui de l'affectation qui utilise le symbole `:=`

<code>let a = 2 ;;</code>	<code>let a = 5 ;;</code>
<code>(* a : int = 2 *)</code>	<code>(* a : int = 5 *)</code>
<code>let b = a + 4 ;;</code>	<code>b ;;</code>
<code>(* b : int = 6 *)</code>	<code>(* - : int = 6 *)</code>

Cet exemple montre que la valeur de `b` reste fixe, égale à celle du moment de la déclaration.

Les identificateurs déclarés globalement s'ajoutent à l'environnement de travail qu'on peut donc enrichir sans cesse.


### 1.5.2 Déclarations simultanées

**Syntaxe** `let ident1 = valeur1 and ident2 = valeur2 and ... and identN = valeurN ;;`  
 ou `let (ident1,ident2,...,identN) = (valeur1, valeur2,...,valeurN) ;;`

### 1.5.3 Déclarations locales d'identificateurs à l'intérieur d'expressions

Il est possible de définir des **identificateurs locaux** à l'intérieur d'une expression lorsque ceux-ci ne sont utiles que dans le calcul de cette expression.

**Syntaxe** `let ident 1 = valeur1 and ...  
in Expression ;;`

 **Important !** Cette phrase, malgré la présence du `let` au début, n'est pas une déclaration, mais est une **expression** Caml de même type que *Expression*.

**Autre syntaxe** `Expression where ident1 = valeur1 and ...;;`


**Remarque.** Cette seconde syntaxe met mieux en évidence qu'il s'agit d'une expression.


```
let f x =
  let pi = 4 *. atan 1. in          (* pi est un identificateur local à la déclaration de f *)
  sin (pi *. x) + cos (pi *. x);;
let pi = 4 *. atan 1. in          (* pi est un identificateur local dans l'expression cos (pi/12) *)
cos (pi /. 12) ;;
```

Il est possible d'emboîter des déclarations locales, ou d'utiliser `and` pour les déclarations locales simultanées.

## 1.6 Séquences d'expressions

En Caml, toute phrase qui n'est ni une définition (ou déclaration), ni une directive de compilation est une **expression**. Il est possible de former des **séquences** d'expressions en les séparant par des **points-virgules** et en les entourant éventuellement de parenthèses ou par `begin` et `end`.

 **Important !** Une séquence d'expressions (`Expr_1;...;Expr_n`) est également une expression qui prend comme valeur celle de la *dernière expression* `Expr_n`.

 **Attention !** Une séquence ne peut donc **pas contenir de déclarations** `let a=...;` et les séquences n'ont d'intérêt pratique que si les expressions (sauf la dernière) réalisent des effets de bord.

**En pratique.** Une séquence est de la forme : `Action_1; ...; Action_p; Expr;`  
 où `Action_1, ..., Action_p` renvoient un résultat de type `unit`.  
 Le résultat de la séquence (et son type) est celui de `Expr`.

```
(print_int 3; print_char `> ; 3) + begin print_int 5; print_char `+`; 5 end ;;
(* 5+3>- : int = 8 *)
```

Cet exemple montre en outre que Caml évalue d'abord le 2<sup>ème</sup> argument de la somme.

## 2 FONCTIONS

Pour Caml, qui se veut un *langage fonctionnel*, les fonctions sont des objets comme les autres et peuvent par conséquent être passées comme arguments à d'autres fonctions (qui prennent alors le nom d'*expressions fonctionnelles*).

### 2.1 Fonction à un seul argument

**Syntaxe** `let f x = Expr ;;`

ou `let f = function x -> Expr ;;` ou `let f(x) = Expr ;;`

Les variables utilisées ici sont *muettes* : la déclaration est valable même si dans l'environnement, *x* s'est vu attribué une valeur auparavant (différence importante avec Maple).

En mode interactif, après déclaration avec un `let` ou à l'appel d'une fonction par son nom seul (sans arguments), Caml fournit une réponse de la forme `t1 -> t2 = <fun>`.

Cela signifie que la fonction prend un argument du type *t1* et renvoie une valeur du type *t2*, mais, contrairement à Maple qui est capable d'afficher sous forme graphique l'expression mathématique de la fonction, Caml se contente de retourner `<fun>` quelle que soit la fonction : il n'est donc pas possible de faire afficher explicitement ce qu'elle réalise.

```
let carré x = x*x ;;
(* carré : int -> int = <fun> *)
let carré(x) = x*x ;;
(* carré : int -> int = <fun> *)
let carré = function x -> x*x ;;
(* carré : int -> int = <fun> *)

carré ;;
(* int -> int = <fun> *)
let exp_rond f = function x-> exp(f x) ;;
(* exp_rond : ('a -> float) -> 'a -> float = <fun> *)
exp_rond cos ;;
(* float -> float = <fun> *)
```

### 2.2 Fonctions de plusieurs variables

Lire d'abord le § 3.2.2 relatif au produit cartésien de types

#### 2.2.1 Fonction définie sur un produit cartésien

Il s'agit en fait d'une fonction à *un seul argument*, celui-ci étant un multiplet.

```
let somme_des_carrés (x,y) = x*x+y*y ;;
(* somme_des_carrés : int * int -> int = <fun> *)
let somme_des_carrés = function x,y -> x*x+y*y ;;
(* somme_des_carrés : int * int -> int = <fun> *)
let somme_des_carrés = function (x,y) -> x*x+y*y ;;
(* somme_des_carrés : int * int -> int = <fun> *)

somme_des_carrés (3,4) ;;
(* int = 25 *)

let division(m,n) = (m/n, m mod n) ;;
(* division : int * int -> int * int = <fun> *)
```

Les parenthèses entourant le multiplet sont ici souvent obligatoires. Voir à ce propos le § 2.2.3 sur la gestion des parenthèses.

```
let somme_des_carrés x,y = x*x+y*y ;;
somme_des_carrés 3,4 ;;
(* Toplevel input: *)
(* >somme_des_carrés 3,4 ;; Il y a eu associativité à gauche *)
(* > ^ (somme_des_carres (3)) , 4 *)
(* This expression has type int, *)
(* but is used with type int * int. *)
```

On peut utiliser des fonctions locales dans une déclaration, par exemple :

```
let somme_des_carrés (x,y) = let carré t = t*t in (carré x) + (carré y) ;;
```

#### 2.2.2 Déclaration curryfiée

**Syntaxe** `let f x y = Expr ;;`

ou `let f = function x -> function y -> Expr ;;` ou `let f = fun x y -> Expr ;;`

**Remarques.**

- Il n'y a ici **ni virgule, ni parenthèses**. Les syntaxes précédentes se généralisent avec  $n$  variables.
- Caml affiche alors comme réponse  $f : t_1 \rightarrow t_2 \rightarrow t_3 = \langle \text{fun} \rangle$ .
- Le symbole  $\rightarrow$  est **associatif à droite** : cela veut dire en fait  $f : t_1 \rightarrow (t_2 \rightarrow t_3) = \langle \text{fun} \rangle$ .
- $f$  est donc une fonction de source  $t_1$  ayant pour valeur une fonction de  $t_2$  vers  $t_3$ .
- $f\ a$  ou  $f(a)$  désigne une fonction de  $t_2$  vers  $t_3$  (*on peut donc passer un seul argument à  $f$* ).
- $f\ a\ b$  ou  $(f\ a\ b)$  ou  $(f\ a)\ b$  ou  $f(a)\ b$  est un objet de type  $t_3$ .
- L'appel  $f(a,b)$  provoque une erreur lorsque  $f$  est déclarée de façon curryfiée.

```
let dérivée f x = let h=0.0001 in (f(x+h)-f(x-h))/(2.0*h);;
```

L'avantage de cette déclaration curryfiée<sup>1</sup> est que **dérivée  $f$**  représente la *fonction* dérivée (approchée) de  $f$ .

**Conversions** entre fonctions curryfiées et non curryfiées

```
let somme_carrés (x,y) =x*x+y*y ;;
(* somme_carrés : int * int -> int = <fun> *)
let somme_carrés_c x y = somme_carrés(x,y) ;;
(* somme_carrés_c : int -> int -> int = <fun> *)

let module x y = sqrt(x*.x+.y*.y) ;;
(* module : float -> float -> float = <fun> *)
let module_dc (x, y) = module x y ;;
(* module_dc : float * float -> float = <fun> *)
```

**2.2.3 Gestion des parenthèses**

D'une façon générale, Caml autorise (et encourage) la suppression de parenthèses dans de nombreux cas. Les parenthèses servent uniquement à indiquer des priorités : dans le doute, en mettre même si elles sont inutiles. Retenir que lors des évaluations, Caml applique **prioritairement les fonctions** et calcule leurs arguments de *droite à gauche*.

**Pour les multiplats.** (seul cas où la virgule est utilisée en Caml)

$x,y$  est identique à  $(x,y)$  de même que  $x_1, \dots, x_n$  est identique à  $(x_1, \dots, x_n)$ .

Ceci n'est plus valable si ce sont des arguments de fonctions : voir exemples au § 2.2.1.

**Pour les arguments de fonctions.**

$f\ x$  est équivalent à  $f(x)$



**Important !** L'associativité a lieu à gauche (comme pour la plupart des opérateurs infixes) :  $f\ x\ y$  est équivalent à  $(f\ x)\ y$



**Attention !** Par défaut, le signe  $-$  est considéré comme **binaire**.

Ainsi lorsqu'il est utilisé comme le symbole unaire, il doit être mis entre parenthèses avec son argument.

**Exemples.**

- Par exemple, en présence de `cos -2`, Caml cherche à effectuer la différence entre `cos` et `2` et renvoie une erreur. La syntaxe correcte est `cos(-2)`.
- `exp cos 3.14` est compris comme étant `(exp cos) 3.14` qui n'a pas de sens. Une syntaxe correcte est `exp(cos 3.14)` ou `exp(cos(3.14))`.
- Si  $f$  est la fonction  $x \mapsto x \times x$ , alors `f 2*3` renvoie 12 : en effet le calcul de `f 2` est prioritaire devant la multiplication.
- Si  $f$  est non curryfiée de type `int * int -> int`, alors la seule façon de la calculer en `(3,4)` est d'écrire `f(3,4)`.
- Si  $f$  est curryfiée de type `int -> int -> int`, alors `f 2*3 4+5` provoque une erreur car Caml essaie d'évaluer `(f 2)*3` qui n'existe pas, alors que `f (2*3) 4+5` est correct et a le sens de `(f (2*3) 4) + 5`

**2.3 Fonctions utilisant le filtrage (ou pattern-matching)**

il s'agit d'une *reconnaissance de motif*.

1. CURRY était un logicien anglais du début du XX<sup>ème</sup> siècle.

## 2. FONCTIONS

### Syntaxe

<pre>let f x = match x with   motif1 -&gt; Expression1   motif2 -&gt; Expression2   .....   motifN -&gt; ExpressionN ;;</pre>	<pre>ou let f = function   motif1 -&gt; Expression1   motif2 -&gt; Expression2   .....   motifN -&gt; ExpressionN ;;</pre>
---	--

### Règles.

- ↪ Tous les *motifs* doivent être du même type  $t_1$ .
- ↪ Toutes les *Expressions* doivent être de même type  $t_2$ .
- ↪ La fonction résultat est alors du type  $t_1 \rightarrow t_2$ .
- ↪ Les motifs sont examinés dans *l'ordre de la déclaration* : en cas de succès de l'un d'eux, les suivants ne sont pas pris en considération. Ainsi les motifs ne sont pas nécessairement disjoints. En revanche, leur « réunion » doit recouvrir tout le type  $t_1$  sous peine d'erreur à l'exécution (*filtrage non exhaustif*).
- ↪ Le motif « joker » `_` (symbole de soulignement) placé en dernier a le sens de « pour tous les cas qui restent ».

### Remarques.

- ↪ Un motif n'est pas une valeur, donc un filtrage n'est pas une comparaison de valeurs. On peut penser le filtrage comme la question « est-ce que la variable est de la forme ... ? » en gardant à l'esprit que les valeurs ne sont pas comparées. On verra au § 4.3 que les gardes permettent de combiner la reconnaissance de motifs et la comparaison de valeurs.
- ↪ L'interpréteur **Caml** détecte lors du typage les filtres inutiles ou incomplets. Il convient donc de lire les messages de l'interpréteur !
- ↪ Il faut se méfier des filtrages emboîtés, et bien penser à délimiter les filtrages intérieurs par **begin** et **end**. On peut trouver des exemples sur la faq en ligne.

**Exemples.** Voyons un premier exemple d'apparence simple :

```
let f x =
  match x with
  | 0 -> print_string "zéro"
  | 1 -> print_string "un"
  | _ -> print_string "beaucoup"
;;
(* f : int -> unit = <fun> *)
```

La compréhension de cette fonction ne pose pas de problème. Attention cependant, `x` n'est pas comparé à la valeur 0, mais à la constante 0 comme l'illustre l'exemple suivant, et qui peut surprendre :

```
let a = 2 and x = 1;;
match x with
| a -> print_string "mêmes motifs"
| _ -> print_string "motifs différents"
;;
(* Entrée interactive: *)
(* > | _ -> print_string "motifs différents";; *)
(* > ^ *)
(* Attention: ce cas de filtrage est inutile. *)
(* mêmes motifs- : unit = () *)
```

Il s'explique car le `a` du filtrage est un motif universel, qui correspond donc à `x`. Le second filtrage est donc inutile, et l'expression produit l'écriture de *mêmes motifs* indépendamment des valeurs de `x` et `a`.

Voici enfin un exemple non trivial où le filtrage n'est pas exhaustif :

```
let est_pair x =
  match (x mod 2) with
  | 0 -> true
  | 1 -> false
;;
(* Entrée interactive: *)
(* > ..match (x mod 2) with *)
(* > | 0 -> true *)
(* > | 1 -> false.. *)
```



```
(* Attention: ce filtrage n'est pas exhaustif. *)
(* est_pair : int -> bool = <fun> *)
```

Il faut compléter la définition par `_ -> failwith "argument non valable"`

## 2.4 Fonctions récursives

Une fonction est dite **récursive** si son nom intervient dans le corps de sa définition.

L'identificateur d'une telle fonction doit être précédé du mot-clé `rec`.

La plupart des fonctions Caml sont récursives et utilisent le filtrage.

**Exemples.** Factorielle et calcul de la somme  $\sum_{i=m}^n f(i)$ .

```
let rec factorielle n = match n with
  | 0 -> 1
  | n -> n * factorielle (n-1) ;;
(* factorielle : int -> int = <fun> *)

let rec sigma f m n =
  if m=n then f m
  else (f m) + (sigma f (m+1) n) ;;
(* sigma : (int -> int) -> int -> int -> int = <fun> *)
sigma (fun x -> x * x) 1 5 ;;
(* - : int = 55 *)
```

## 2.5 Conversion entre opérateurs binaires infixes et préfixes

Un opérateur **infixe** est un opérateur binaire qui s'écrit entre ses arguments (comme `+`, `<`, `^`, `@`, `::`).

Ces opérateurs ne sont pas syntaxiquement considérés par Caml comme des fonctions et ne peuvent donc pas être passés en argument de fonctionnelles.

Pour transformer un opérateur infixe en une fonction (c-à-d **préfixe curryfié**), on le fait précéder du mot-clé `prefix`.

```
(prefix +) ;;
(* - : int -> int -> int = <fun> *) (* équivalent à add_int *)
```

Inversement, on peut transformer un opérateur binaire préfixe en un opérateur infixe avec `#infix` suivi du nom entre guillemets (de type `string`) de cet opérateur.

```
let o f g = function x -> f(g(x)) ;;
(* o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun> *) (* Composition des fonctions *)
#infix "o";;
(succ o succ o succ) 3 ;;
(* - : 6 *)
```

Si `T` est un opérateur infixe dont la forme préfixe est de type `'a -> 'b -> 'a` comme c'est le cas pour la plupart des opérateurs infixes, (excepté les opérateurs de comparaison booléens), on pourra former des expressions *itérées non parenthésées* du type `a T b1 T b2 T ... T bn` qui représente `((a T b1) T b2) ... T bn` (associativité à gauche). La forme préfixe équivalente nécessiterait, elle, des parenthèses.

On annule l'effet de `#infix "..."` avec `#uninfix "..."`.

On peut déclarer directement avec `prefix` des opérateurs infixes à condition que son identifiant n'utilise que des caractères non alphanumériques comme `+`, `-`, `*`, `/`, `%`, `$`, ...

```
let prefix => p q = not p or q ;;
(* => : bool -> bool -> bool = <fun> *)
```

## 3 LES TYPES DE DONNÉES PRÉDÉFINIS

Un **type** en Caml correspond intuitivement à un **ensemble** en mathématique.  
La notation ``a`, ``b` etc. désigne n'importe quel type.

### 3.1 Les types élémentaires

#### 3.1.1 Le type unit

Ce type ne contient qu'un seul objet noté `()` appelé « rien ». Il est utilisé :

- soit par les fonctions dites « sans argument » de type `unit -> `a`,
- soit pour les fonctions à effets de bords (appelées parfois *procédures*).

Ces fonctions sont des fonctions ayant une action sur le contexte (modification de la mémoire de l'ordinateur, affichage à l'écran, tracé dans une fenêtre graphique). Elles sont de type ``a -> unit`, donc renvoient `()`, c'est-à-dire « rien », l'unique élément de type `unit`.

```
print_int 3;;          quit;;          print_newline ();;
(* 3- : unit = () *)   (* - : unit -> unit = <fun> *)   (*
(* - : unit = () *)
```

#### 3.1.2 Les nombres entiers (int)

Le type `int` est celui des nombres entiers relatifs qui sont implémentés sur un mot de 4 octets (avec 31 bits utilisés)<sup>2</sup> : ainsi un nombre entier est compris entre  $-2^{30}$  et  $2^{30} - 1$  et est défini modulo  $2^{31}$ .

On dispose avec le type `int` de :

- cinq opérateurs binaires (infixes) `+ - * / mod` qui renvoient tous un entier.
- l'opérateur unaire `-` de changement de signe.
- six opérateurs binaires (infixes) de comparaison `= < <= > >= <>` qui renvoient un booléen.
- les deux valeurs `min_int` et `max_int` qui sont respectivement le plus grand et plus petit entier machine.

```
11 / 4 ;;          11 mod 4 ;;          11 > 4 ;;
(* - : int = 2 *)   (* - : int = 3 *)   (* - : bool = true *)
```

La multiplication et la division sont prioritaires devant l'addition et la soustraction.

L'opérateur modulo `mod` est prioritaire devant les quatre autres opérateurs binaires.

Les quatre opérations usuelles sur les entiers peuvent aussi être réalisées avec quatre fonctions de type `int -> int -> int` correspondant à un opérateur préfixé. Ce sont : `add_int` `sub_int` `mult_int` `div_int`

On dispose en outre des fonctions suivantes :

<code>minus_int</code>	<code>int -&gt; int</code>	$k \mapsto -k$	<code>abs</code>	<code>int -&gt; int</code>	$k \mapsto  k $
<code>succ</code>	<code>int -&gt; int</code>	$k \mapsto k + 1$	<code>min</code>	<code>int -&gt; int -&gt; int</code>	$m \mapsto (n \mapsto \inf\{m, n\})$
<code>pred</code>	<code>int -&gt; int</code>	$k \mapsto k - 1$	<code>max</code>	<code>int -&gt; int -&gt; int</code>	$m \mapsto (n \mapsto \sup\{m, n\})$

Il existe enfin des fonctions de conversion entre entiers et chaînes de caractères :

<code>string_of_int</code>	<code>int -&gt; string</code>	Convertit un entier écrit sous sa forme décimale en la chaîne dont les caractères sont les chiffres (précédés éventuellement d'un "-").
<code>int_of_string</code>	<code>string -&gt; int</code>	Convertit une chaîne de caractères formée d'un "-" éventuellement et de chiffres en un nombre entier modulo $2^{31}$ .

#### 3.1.3 Les nombres flottants (float)

Le type `float` permet de représenter des nombres réels avec au plus 12 chiffres significatifs avec nos machines. L'écriture de ces nombres contient nécessairement un point décimal ou un `e` (ou `E`).

Exemples : `2.`, `-123.456789`, `1.23456789e-008` ou `123E50`.

Les opérateurs usuels se notent : `+. -. *. /. **.` et `= < <= > >= <>`

Les cinq opérations usuelles sur les nombres flottants peuvent aussi être réalisées avec les cinq fonctions de type `float -> float -> float` correspondant à un opérateur préfixé. Ce sont :

`add_float` `sub_float` `mult_float` `div_float` `power`

On dispose en outre des fonctions usuelles suivantes de type `float -> float` :

2. Pour les processeurs 64 bits, le nombre est bien-sûr défini modulo  $2^{63}$

abs_float	sqrt	sin	cos	tan	acos	asin	atan	exp	log
-----------	------	-----	-----	-----	------	------	------	-----	-----

`log` désigne en fait le logarithme népérien.

Les arguments des fonctions trigonométriques sont en radians.

Il existe enfin des fonctions de conversion entre flottants, entiers et chaînes de caractères :

<code>float_of_int</code>	<code>int -&gt; float</code>	<code>float_of_int 3;;</code> $\xrightarrow{\text{Cam1}}$ <code>- : float = 3.0</code>
<code>int_of_float</code>	<code>float -&gt; int</code>	<code>int_of_float (-4.153);;</code> $\xrightarrow{\text{Cam1}}$ <code>- : int = -4</code>
<code>string_of_float</code>	<code>float -&gt; string</code>	Convertit un flottant en une chaîne.
<code>float_of_string</code>	<code>string -&gt; float</code>	Convertit une chaîne en un flottant si cela a un sens.



**Attention !** Pour les flottants négatifs, `int_of_float` n'est pas la partie entière

### 3.1.4 Les caractères (char)

Ils sont de type `char` et sont représentés entre deux accents graves (*backquotes*) comme ``a``.

Il y a quatre caractères spéciaux qui sont `\` (backslash), ``` (backquote), le retour-chariot et la tabulation.

Ils se notent respectivement : `\\` \n` \t``

Les opérateurs de comparaison selon l'ordre (total) ASCII sont `= < <= > >= <>`

Les fonctions de conversion suivantes utilisent le codage ASCII des caractères :

<code>int_of_char</code>	<code>char -&gt; int</code>	<code>char_of_int</code>	<code>int -&gt; char</code>	<code>char_for_read</code>	<code>char -&gt; string</code>
--------------------------	-----------------------------	--------------------------	-----------------------------	----------------------------	--------------------------------

```
char_of_int 65 ;;           `065` ;;
(* - : char = `A` *)      (* - : char = `A` *)
```

codes ASCII intéressants

9 \t	tabulation	10 \n	retour chariot			
32	33 !	34 "	35 #	36 \$	37 %	38 &
40 (	41 )	42 *	43 +	44 ,	45 -	46 .
48 0	49 1	50 2	51 3	52 4	53 5	54 6
56 8	57 9	58 :	59 ;	60 <	61 =	62 >
64 @	65 A	66 B	67 C	68 D	69 E	70 F
72 H	73 I	74 J	75 K	76 L	77 M	78 N
80 P	81 Q	82 R	83 S	84 T	85 U	86 V
88 X	89 Y	90 Z	91 [	92 \	93 ]	94 ^
96 `	97 a	98 b	99 c	100 d	101 e	102 f
104 h	105 i	106 j	107 k	108 l	109 m	110 n
112 p	113 q	114 r	115 s	116 t	117 u	118 v
120 x	121 y	122 z	123 {	124	125 }	126 ~

### 3.1.5 Les booléens (bool)

Ce type, constitué de deux constantes appelées `true` et `false`, est défini comme : `type bool = true | false` (voir § 3.3.1).

**Prédicats.** C'est le nom donné à toute fonction à valeur booléenne (de type `'a -> bool`).

**Opérateurs sur les booléens.** On dispose avec ce type de trois opérateurs (on dit plutôt des connecteurs) :

↪ deux opérateurs binaires (infixes) ayant chacun deux notations possibles :

« et » noté `&` (ou `&&`)    « ou » noté `or` (ou `||`)

↪ d'un opérateur unaire

« non » noté `not`

Le 2<sup>ème</sup> prédicat des deux connecteurs `&` et `or` n'est pas toujours évalué, cela dépend de l'évaluation du 1<sup>er</sup> prédicat :

- dans `p & q`, `q` n'est évalué que si `p` est vrai (cf `if p then q else false`).
- dans `p or q`, `q` n'est évalué que si `p` est faux (cf `if p then true else q`).



**Attention !** `and` est utilisé en Cam1 (avec `let` par exemple), mais n'a rien à voir avec le connecteur « et ».

### 3. LES TYPES DE DONNÉES PRÉDÉFINIS

#### Opérateurs de comparaison.

- ↪ `=` `<` `<=` `>` `>=` `<>` opérateurs polymorphes (s'appliquent aux entiers, flottants, chaînes...)
- ↪ l'égalité *structurelle* notée `=` applicable pour vérifier l'égalité de deux expressions de n'importe quel type `a` : les deux valeurs sont identiques, mais peuvent être stockées à deux adresses différentes.
- ↪ l'inégalité *structurelle* notée `<>` : la négation de la précédente.
- ↪ l'égalité *physique* notée `==` applicable pour vérifier l'égalité des adresses de deux expressions de n'importe quel type.
- ↪ l'inégalité *physique* notée `!=` : la négation de la précédente.

#### Exemple.

```
let a = [1;2;3] and b=[1;2;3];;
a=b;;
(* - : bool = true *)
a==b;;
(* - : bool = false *)
```

#### Fonctions de comparaison.

<code>eq_int</code>	<code>neq_int</code>	<code>le_int</code>	<code>ge_int</code>	<code>lt_int</code>	<code>gt_int</code>	<code>(type int -&gt; int -&gt; bool)</code>
<code>eq_float</code>	<code>neq_float</code>	<code>le_float</code>	<code>ge_float</code>	<code>lt_float</code>	<code>gt_float</code>	<code>(type float -&gt; float -&gt; bool)</code>
<code>eq_string</code>	<code>neq_string</code>	<code>le_string</code>	<code>ge_string</code>	<code>lt_string</code>	<code>gt_string</code>	<code>(type string -&gt; string -&gt; bool)</code>

## 3.2 Les types composés

### 3.2.1 Les types fonctionnels

Ils ont déjà été évoqués précédemment : `t1 -> t2` est le type des fonctions de source `t1` et de but `t2`.

### 3.2.2 Les produits cartésiens

Si `t1, ..., tn` sont des types quelconques (éventuellement distincts), alors `t1*...*tn` est le type des *n*-uplets (ou *multi-plets*)  $(x_1, \dots, x_n)$  avec  $x_i$  du type `ti`.

À noter que c'est le seul cas en Caml où intervient une virgule comme séparateur et que les parenthèses extérieures peuvent souvent être supprimées.

Les fonctions prédéfinies disponibles uniquement sur les *couples* (on dit plutôt *paires* en informatique) sont :

<code>fst</code>	<code>'a*'b -&gt;'a</code>	renvoie le premier élément d'un couple	<code>snd</code>	<code>'a*'b -&gt;'a</code>	renvoie le second élément
------------------	----------------------------	--	------------------	----------------------------	---------------------------

**Remarque.** L'instruction `let (_,_,x,_) = ...` permet de faire une liaison, et de lier `x` à la 3<sup>ème</sup> composante du quadruplet.

### 3.2.3 Les listes

Une liste est une suite finie (de longueur variable contrairement aux tableaux) d'éléments tous de même type `'a`.

Le type de la liste est alors `'a list` et celle-ci est affichée par Caml sous la forme `[e1; e2; ...; en]`.

**Exemple.** `[0; 1; 4; 9; 16]` est une liste de type `int list`.

La définition d'une liste est **récursive** et utilise le **constructeur de liste** noté `::` (qu'on lit *quatre points* ou *cons*). Elle peut être modélisée par

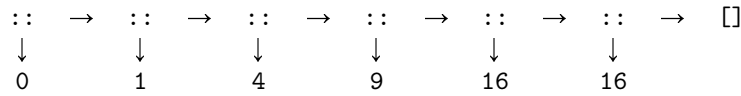
```
type 'a list =
| Nil
| Cons of 'a * 'a list ;;
```

```
type 'a list = [] | prefix :: of 'a * 'a list
```

- il existe une liste vide notée `[]` (*nil*)
- si `t` est un élément de type `a` et `q` une liste de type `a list`, alors la liste dont le premier élément (la *tête*) est `t` et dont les suivants (la *queue* ou le *reste*) sont ceux de la liste `q` se note `t :: q`

L'opérateur `::` est forcément « associatif à droite ».

La liste précédente correspond mathématiquement au couple  $(0, (1, (4, (9, (16, \emptyset))))$  et à l'arbre binaire:



La liste de l'exemple précédent peut être entrée de plusieurs façons, dont :

```
[0;1;4;9;16];;
0::1::4::9::16::[];;      (* c'est la même liste *)
0::1::4::9::16::[];;      (* c'est encore la même liste *)
```

À part l'élément de tête, on n'a pas directement accès aux autres éléments d'une liste comme à ceux d'un tableau et la plupart des fonctions opérant sur les listes sont *récurives*.

La **concaténation** de listes s'effectue grâce à l'opérateur infixe (associatif) noté `@`, qui se lit « *append* » et qui admet `[]` pour élément neutre. Écrire une fonction qui effectue cette concaténation est un exercice très classique.

La **tête** (*head*) et la **queue** (*tail*) de la liste peuvent être extraites grâce aux deux fonctions suivantes :

```
hd : 'a list -> 'a et tl : 'a list -> 'a list
```



**Attention !** On n'utilise que rarement ces fonctions. En particulier, lors de l'écriture de fonctions récurives sur les listes, on utilise le filtrage :

```
match liste with
| [] -> expr1
| t :: q -> expr2
```

**Exemples.** Définition de fonctions sur les listes utilisant le filtrage :

```
let hd l = match l with
| t :: _ -> t
| _ -> failwith "pas de tête";;

let rec list_length = function
| [] -> 0
| t :: q -> 1 + (list_length q);;

let rec rev = function
| [] -> []
| t :: q -> (rev q) @ [t];;
```

**Remarque.** Cette dernière fonction `rev` n'est pas très efficace !

**Fonctions prédéfinies sur les listes.** Toutes les fonctions doivent savoir être redéfinies. On peut trouver une liste complète des fonction prédéfinies dans la documentation en ligne. Donnons cependant les définitions suivantes :

- `list_length` `'a list -> int`  
Renvoie la longueur (nombre d'éléments) de la liste.
- `rev` `'a list -> 'a list`  
Renvoie la liste « renversée ».  
`rev [5;8;12];;`  $\xrightarrow{\text{Cam1}}$  `- : int list = [12; 8; 5]`
- `prefix @` `'a list -> 'a list -> 'a list`  
Concatène deux listes.  
`[3] @ [1; 4; 1; 6];;`  $\xrightarrow{\text{Cam1}}$  `- : int list = [3; 1; 4; 1; 6]`
- `map` `('a -> 'b) -> 'a list -> 'b list`  
Renvoie la liste des images par une fonction des éléments d'une liste.  
`let div3 n = (n mod 3) = 0;;`  $\xrightarrow{\text{Cam1}}$  `div3 : int -> bool = <fun>`  
`map div3 [5;8;12];;`  $\xrightarrow{\text{Cam1}}$  `- : bool list = [false; false; true]`
- `do_list` `('a -> 'b) -> 'a list -> unit`  
Idem, mais ne renvoie que `()` ! Surtout utile avec des effets de bord.  
`do_list f [e1; ...; en];;` effectue la séquence `begin f e1; ...; f en; () end`.  
`let écrit a = print_int a; print_char ` `;;`  $\xrightarrow{\text{Cam1}}$  `écrit : int -> unit = <fun>`  
`do_list écrit [5;8;12];;`  $\xrightarrow{\text{Cam1}}$  `5 8 12 - : unit = ()`
- `mem` `'a -> 'a list -> bool`  
Teste si un élément est dans la liste (au sens de l'égalité structurelle =).  
`mem 8 [5;8;12];;`  $\xrightarrow{\text{Cam1}}$  `- : bool = true`  
`mem 3 [5;8;12];;`  $\xrightarrow{\text{Cam1}}$  `- : bool = false`

### 3. LES TYPES DE DONNÉES PRÉDÉFINIS

**Tri et fusion de listes** Le module « sort » (non chargée par défaut) contient deux fonctions : l'une de tri (`sort`) et l'autre de fusion (`merge`) de deux listes déjà triées (fonctions qu'il faut savoir redéfinir).

- `sort` (`'a -> 'a -> bool`)  $\rightarrow$  `'a list -> 'a list = <fun>`  
`sort (prefix <=) [2;5;8;1;9] ;;`  $\xrightarrow{\text{Caml}}$  `- : int list = [1; 2; 5; 8; 9]`
- `merge` (`'a -> 'a -> bool`)  $\rightarrow$  `'a list -> 'a list -> 'a list = <fun>`

## 3.3 Types définissables par l'utilisateur

### 3.3.1 Les types somme

Ils correspondent au « ou exclusif » .

Les objets de ce type ne sont pas modifiables : ils sont en lecture seule.

#### Types énumérés (à constructeurs constants)

```
type Nom_du_type = Constr_1 | Constr_2 | ... | Constr_n;;
```

Les `Constr_i` sont appelés des **constructeurs** qui sont ici constants.

L'usage est de leur donner un nom commençant par une majuscule.

Un exemple prédéfini est : `type bool = true | false` où `true` et `false` sont deux constantes du langage Caml.

```
type couleur = Rouge | Vert | Bleu ;;           let c = Vert ;;           c = Vert ;;
(* Type couleur defined. *)                     (* c : couleur = Vert *)         (* - : bool = true *)
```

Le type `couleur` se compose de trois objets : `Rouge`, `Vert`, `Bleu` sont devenues des *constantes* du langage qui ce sont ajoutées à l'environnement et seront accessibles durant toute la session de travail.

#### Types somme à constructeurs non constants

Ils permettent de réunir des types disjoints `t1, ..., tn` en un seul type.

```
type Nom_du_type = Constr_1 of t1 | ... | Constr_n of tn;;
```

Les objets de ce nouveau type se notent `Constr_i(x)` ou `Constr_i x` où `x` de type `ti`.

L'intérêt d'un tel type est de pouvoir lui appliquer facilement le filtrage à l'aide des constructeurs.

```
type numéric = Entier of int | Réel of float ;;           type complex = C of float*float ;;
(* Type numéric defined. *)                             (* Type complex defined. *)
let a = Entier 15 ;;                                     let prefix +: (C(a,b)) (C(c,d))=C(a+.c,b+.d)
(* a : numéric = Entier 15 *)                             (* +: : complex -> complex -> complex = <fun> *)
let b = Réel(3.14) ;;
(* b : numéric = Réel 3.14 *)
est_entier = function
  | Entier n -> true
  | _       -> false ;;
(* est_entier : numéric -> bool = <fun> *)
est_entier a ;;
(* - : bool = true *)
print_numéric = function
  | Entier n -> print_int(n)
  | Réel x   -> print_float(x) ;;
(* print_numéric : numéric -> unit = <fun> *)
let add_numéric x y = match (x,y) with
  | (Entier n , Entier m) -> Entier(add_int m n)
  | (Réel u , Réel v)     -> Réel(add_float u v)
  | (Entier n, Réel u)     -> Réel(add_float (float_of_int n) u)
  | (Réel u, Entier n)     -> Réel(add_float (float_of_int n) u) ;;
(* add_numéric : numéric -> numéric -> numéric = <fun> *)
add_numéric a b ;;
(* - : numéric = Réel 18.14 *)
```

#### Types somme à constructeurs mixtes

Il est possible de définir des types somme contenant à la fois des constructeurs constants et non constants.

```
type réel_étendu = Plus_infini | Moins_infini | Réel of float ;;
(* Type réel_étendu defined. *)
```

### Types somme rékursifs

Exemple d'un arbre binaire dont les feuilles identifiées par le constructeur F sont des entiers et dont les nœuds identifiés par N ont pour étiquette un caractère (symbolisant un opérateur).

```
type arbre =
  | F of int
  | N of arbre * char * arbre ;;
(* Type arbre defined. *)
let rec eval x =
  match x with
  | F n -> n
  | N(a, '+', b) -> (eval a) + (eval b)
  | N(a, '-', b) -> (eval a) - (eval b)
  | N(a, '*', b) -> (eval a) * (eval b)
  | N(a, '/', b) -> (eval a) / (eval b)
  | _ -> failwith "Erreur";; (* on déclenche une exception *)
(* eval : arbre -> int = <fun> *)
let a = N(N(F(2), '+', F(3)), '*', N(F(7), '-', F(4))) ;;
(* a : arbre = N (N (F 2, '+', F 3), '*', N (F 7, '-', F 4)) *)
eval a ;;
(* int = 15 *)
```

Autre définition possible d'un tel arbre mélangeant les types somme et produit et la récursivité « croisée » :

```
type arbre = Vide | Noeud of Branche
and Branche = {Op : char; Fils_g : arbre; Fils_d : arbre};;
```

Exemple de typage possible pour une expression arithmétique ne contenant que des entiers et des opérateurs binaires :

```
type expr_arith =
  | Const of int
  | Addi of opérandes
  | Diff of opérandes
  | Mult of opérandes
  | Quot of opérandes
and opérandes = {Op1 : expr_arith; Op2 : expr_arith};;
```

### 3.3.2 Les types produit (ou enregistrements)

Ils correspondent au « et » logique : ce sont des types de données vérifiant simultanément plusieurs propriétés.

Il s'agit en fait d'une variante améliorée du type produit cartésien des types  $t_1, \dots, t_n$  dans lequel les composantes (appelées **champs**) ont un nom (appelé **étiquette**), ce qui permet de réaliser facilement le filtrage, les étiquettes jouant le rôle de motifs.

**Syntaxe** `type Nom_du_type = { Etiqu_1:t1; Etiqu_2:t2; ...; Etiqu_n:tn };;`

Le couple  $Etiqu_i:t_i$  s'appelle le  $i^{\text{ème}}$  **champ**.

**Déclaration d'une valeur de ce type** `let Nom = { Etiqu_1 = val1; ...; Etiqu_n = valn };;`

L'ordre dans lequel on déclare les champs n'a pas d'importance.

Les objets de ce type ne sont pas modifiables : ils sont en lecture seule. (Voir à ce titre le § 3.4 sur les types mutables)

**Accès à la valeur du  $i^{\text{ème}}$  champ** grâce la notation pointée `Nom.Etiqu_i`.

```
type régime = Interne | Externe | Interne_Externé ;;
(* Type régime defined. *)
type élève = {Nom : string ; Age : int ; Qualité : régime}
(* Type élève defined. *)
let Lemec = {Age=20; Qualité=Interne; Nom = "Le Mec"};;
```

### 3. LES TYPES DE DONNÉES PRÉDÉFINIS

```
(* Lemec : élève = {Nom = "Le Mec"; Age = 20; Qualité = Interne} *)
Lemec.Qualité=Interne ;;
(* - : bool = true *)

type cercle = {Centre:int*int; Rayon:int} ;;
(* Type cercle defined. *)
let gamma = {Centre = 5,6; Rayon = 3}
(* gamma : cercle = {Centre = 5, 6; Rayon = 3} *)
gamma.Rayon ;;
(* - : int = 3 *)

type rationnel = {Num : int; Den : int} ;;
(* Type rationnel defined. *)
let add_rationnel r s = {Num = r.Num * s.Den + r.Den * s.Num; Den=r.Den*s.Den};;
(* add_rationnel : rationnel -> rationnel -> rationnel = <fun> *)

type complexe = {Re:float; Im:float}
(* Type complexe defined. *)
let z1 = {Re = 3.0; Im = 4.0} ;;
(* z1 : complexe = {Re = 3.0; Im = 4.0} *)
let module z = sqrt(z.Re *. z.Re +. z.Im *. z.Im) ;;
(* module : complexe -> float = <fun> *)
module z1 ;;
(* - : float = 5.0 *)
let add_complexe z z' = {Re = z.Re +. z'.Re; Im = z.Im +. z'.Im} ;;
(* add_complexe : complexe -> complexe -> complexe = <fun> *)
add_complexe z1 z1 ;;
(* complexe = {Re = 6.0; Im = 8.0} *)
```

#### 3.3.3 Types polymorphes (ou paramétrés)

Ce sont des types dont la déclaration contient des variables de type 'a, 'b, ...

```
type ('a,'b) paire = {Premier : 'a; Second : 'b} ;;
(* Type paire defined. *)
type ('a,'b) un_ou_deux = Singleton of 'a | Couple of 'a * 'b ;;
(* Type un_ou_deux defined. *)
let elt = Singleton(3) ;;
(* elt : (int, 'a) un_ou_deux = Singleton 3 *)
```

Il est possible de redéfinir les listes et les fonctions associées en utilisant le constructeur (préfixe) noté ici C.

```
type 'a liste = Nil | C of 'a * 'a liste ;;
(* Type liste defined. *)
let maliste = C(7,C(5,C(3,Nil))) ;;
(* maliste : int liste = C (7, C (5, C (3, Nil))) *)
let tête l = match l with
| C(t,_) -> t
| _ -> failwith "Vide" ;;
(* tête : 'a liste -> 'a = <fun> *)
tête maliste ;;
(* - : int = 7 *)
```

#### 3.3.4 Renommer des types

Il est possible de définir des types abréviations, notamment à des produits cartésiens de types en utilisant `==`

```
type intbool == int -> bool ;;
(* Type intbool defined. *)
type complexe == float * float ;;
(* Type complexe defined. *)
```



### 3.3.5 Imposer le type des fonctions

Il est possible d'utiliser des types abréviations pour typer les fonctions, en utilisant `(var : type)` comme illustré dans l'exemple suivant :

```
type ensemble == int list;;
(* Type ensemble defined. *)

let card (ensX : ensemble) =
  let rec aux l n =
    (* retourne n + card l *)
    match l with
    | [] -> n
    | t :: q -> aux q (n+1)
  in
  aux ensX 0;;
(* ce n'est plus card : 'a list -> int = <fun> *)
(* mais card : ensemble -> int = <fun> *)
```

## 3.4 Les types mutables

Les variables de ces types sont modifiables en totalité ou partiellement.

Dans la déclaration de leur type, doit figurer le mot-clé **mutable** devant certains motifs.

Les modifications de ces variables sont possibles grâce à des fonctions de type `'a -> unit` qui vont écrire dans une zone de la mémoire (effet de bord). Ces fonctions devront donc en général être placées dans une séquence.

Il existe quelques fonctions prédéfinies comme `:=` (pour les références) et `<-` (pour les autres types).

### 3.4.1 Les références

Le type `'a ref` permet d'obtenir des **variables**, modifiables par **affectation** avec le symbole `:=`, courant dans la plupart des langages informatiques.

Si `'a` est un type, alors le type `'a ref` est celui des références à ce type :

`type 'a ref = ref of mutable 'a`

Il s'agit donc d'un type somme à un seul cas : `ref` est donc le nom du constructeur de ce type.

#### Fonctions prédéfinies

- Déclaration d'une variable : `let var = ref val` ou `let var = ref (val)`  
C'est le type de `val` qui détermine celui de `var`. `var` est en fait un **pointeur** sur `val`. Une variable définie par référence est donc toujours *initialisée* au moment de sa déclaration.
- Lecture d'une variable : on utilise l'opérateur unaire `!` de **déférencement** : `!var` retourne la valeur pointée *valeur*.
- Affectation d'une variable : on utilise l'opérateur binaire `:=` (dont le préfixe est de type `'a ref -> 'a -> unit`) qui réalise donc un effet de bord en allant modifier une zone-mémoire.
- Incrémentation d'une variable entière : `incr int ref -> unit` (cf fonction `r -> r := succ !r`).
- Décrémentement d'une variable entière : `decr int ref -> unit` (cf fonction `r -> r := pred !r`).

```
let compt = ref 1 ;;
(* compt : int ref = ref 1 *)
compt := !compt + 2 ;;
(* - : unit = () (Caml n'affiche pas la modification) *)
!compt ;;
(* - : int = 3 *)
compt ;;
(* - : int ref = ref 3 (Bien faire la différence entre les deux) *)

let fact n =
  (* Factorielle itérative *)
  let p = ref 1 in
  for i = 2 to n do
    (* p fait réf. à 1 x 2 x ... x (p-1) *)
    p := !p * i
  done;
  !p
;;
```

### 3. LES TYPES DE DONNÉES PRÉDÉFINIS

```
(* fact : int -> int = <fun> *)

let sigma f m n =                                (* Fonction sigma itérative *)
  let s = ref 0 in
  for i = m to n do                               (* s fait réf. à f(m) + f(m+1) + ... + f(i-1) *)
    s := !s + (f i)
  done;
  !s;;
(* sigma : (float -> float) -> int -> int -> float = <fun> *)
```



Si **a** est une variable définie par référence et si on déclare **let b = a**, alors **a** et **b** sont deux variables qui pointent sur la même adresse-mémoire : en conséquence toute modification de **a** avec **:=** s'effectue aussi sur **b** et vice-versa.

#### 3.4.2 Les tableaux (ou vecteurs)

Un tableau est une suite finie (de taille fixée lors de la déclaration) d'éléments **tous de même type 'a**.

Le type du tableau est alors **'a vect** et celui-ci peut être entré au clavier et affiché par Caml sous la forme :

`[|e_0; e_1; e_2; ...; e_p|]`.

Les composantes d'un tableau sont rangées dans des zones mémoires consécutives (contrairement aux listes).

Un tableau n'est pas redimensionnable une fois créé, par contre, ses composantes sont *mutables* : elles peuvent donc être modifiées avec l'opérateur `[<-]`.

La **numérotation des composantes commence à 0** : si **t** est un tableau de longueur **n**, alors ses composantes s'écrivent :

**t.(0)**, **t.(1)**, ..., **t.(n-1)** et la modification de la composante de numéro **i** se fait avec : `t.(i) <- valeur` (de type **'a vect -> unit**) : Caml *n'affiche rien*.



Si **v** est un tableau et que l'on déclare **let w = v**, alors **v** et **w** sont deux noms donnés au *même tableau* : toute modification effectuée sur l'un se produit aussi sur l'autre (comme avec Maple). La fonction **copy\_vect** permet de recopier **v** dans **w**. On trouvera plus d'explication sur la faq en ligne.

**Fonctions prédéfinies sur les tableaux** Toutes les fonctions doivent savoir être redéfinies. On peut trouver une liste complète des fonction prédéfinies dans la documentation en ligne. Donnons cependant les définitions suivantes :

- `vect_length` **'a vect -> int**  
Renvoie la taille (nombre d'éléments) du tableau.
- `make_vect` **int -> 'a -> 'a vect**  
`make_vect n x` renvoie un tableau de longueur **n** dont tous les éléments sont égaux physiquement à **x**.
- `concat_vect` **'a vect -> 'a vect -> 'a vect**  
`concat_vect v1 v2` renvoie un tableau obtenu par concaténation des deux vecteurs **v1** et **v2**.
- `sub_vect` **'a vect -> int -> int -> 'a vect**  
`sub_vect v deb lg` renvoie un tableau de longueur **lg** extrait de **v** à partir du numéro **deb**.
- `copy_vect` **'a vect -> 'a vect**  
Renvoie une copie du vecteur.
- `make_matrix` **int -> int -> 'a -> 'a vect vect = <fun>**  
`make_matrix n p x` crée une matrice de dimension (**n,p**) initialisée physiquement avec la valeur **x**. On accède à l'élément d'indice (**i,j**) avec **a.(i).(j)**.

**Cas des tableaux à deux dimensions** La seule manière de procéder est de créer un tableau de tableaux (les tableaux de Caml sont unidimensionnels : ce ne sont que des vecteurs au sens mathématique).

Si l'on écrit naïvement la création d'un tableau à plusieurs dimensions, le résultat n'est pas celui attendu, car on crée sans le vouloir un partage physique des lignes du tableau :

```
let matrice_2_3 = make_vect 2 (make_vect 3 0);;
(* matrice_2_3 : int vect vect = [| [|0; 0; 0]; [|0; 0; 0]|] *)
matrice_2_3.(0).(0) <- 1;;
(* - : unit = () *)
matrice_2_3;;
(* - : int vect vect = [| [|1; 0; 0]; [|1; 0; 0]|] *)
```

En effet l'allocation d'un tableau consiste à calculer la valeur d'initialisation et à la mettre dans chaque case du tableau (c'est pourquoi la ligne de la matrice qui est allouée par l'expression `(make_vect 3 0)` est unique et physiquement partagée par toutes les cases du tableau **matrice\_2\_3**).

Solution : utiliser la primitive `make_matrix`, qui fabrique directement une matrice dont les cases sont remplies par la valeur d'initialisation fournie. L'autre solution consiste à écrire le programme qui allouera explicitement une nouvelle ligne pour chaque ligne du tableau. Par exemple :

```
let matrice_n_m n m x =
  let resultat = make_vect n (make_vect m x) in
  for i = 1 to n - 1 do
    resultat.(i) <- make_vect m x
  done;
  resultat;;
(* matrice_n_m : int -> int -> 'a -> 'a vect vect = <fun> *)
```

De même, la fonction `copy_vect` ne donne pas le résultat attendu pour des matrices : il faut écrire une fonction de copie qui copie explicitement le contenu de toutes les lignes de la matrice :

```
let copy_matrix m =
  let l = vect_length m in
  if l = 0 then m else
  let result = make_vect l m.(0) in
  for i = 1 to l - 1 do
    let coli = copy_vect m.(i) in
    result.(i) <- coli
  done;
  result;;
(* copy_matrix : 'a vect vect -> 'a vect vect = <fun> *)
```

### 3.4.3 Les chaînes de caractères (string)

Ce sont des suites (ou tableaux) de caractères, **numérotées à partir de 0** et délimitées entre des guillemets `"` (*quotes*), de type `string` comme `"Salut"` ou `"abc\"cde"` (cette dernière représente la chaîne « `abc" cde` »).

On peut bien sûr inclure des caractères spéciaux en les écrivant comme indiqué ci-dessus, sans oublier le `\`.

La chaîne vide est notée `"`

On pourra modifier des caractères d'une chaîne puisqu'il s'agit de tableaux (*voir les types mutables*, § 3.4).

Les opérateurs binaires (infixes) sur les chaînes sont :

l'opérateur de **concaténation** : `^` les opérateurs booléens de comparaison `=` `<` `<=` `>` `>=` `<>`

**Fonctions disponibles sur les chaînes de caractères.** On peut trouver une liste complète des fonction prédéfinies dans la documentation en ligne. Rappelons que la numérotation commence à 0, et donnons les définitions suivantes :

- `string_length` `string -> int`  
Renvoie la longueur de la chaîne.
- `s.[n]` retourne le  $n^{\text{ème}}$  caractère de la chaîne `s`.
- `s.[n] <- c` remplace le  $n^{\text{ème}}$  caractère de `s` par `c`.
- `make_string` `int -> char -> string`  
`make_string n 'a'` renvoie une chaîne de longueur `n` dont tous les éléments sont égaux physiquement à `'x'`.
- `sub_string` `string -> int -> int -> string`  
`sub_string s n lg` extrait de `s` à partir du caractère d'indice `n` la sous-chaîne de longueur `lg`.
- `replace_string` `string -> string -> int -> unit`  
`replace_string s1 s2 n` copie `s2` dans `s1` à partir du caractère numéro `n` de `s1`.
- `print_string` `string -> unit`  
Affiche une chaîne à l'écran.

```
sub_string "éternelles" 5 4 ;;
(* - : string = "elle" *)
let ch="banque" ;;
(* ch : string = "banque" *)

replace_string ch "rais" 1 ;;
(* - : unit = () *)
ch ;;
(* - : string = "braise" *)
```

### 3.4.4 Les enregistrements à champs variables

Il est possible de définir des types produit dont certains champs pourront être modifiés grâce à `<-` à condition de les faire précéder du mot-clé `mutable` dans leur définition.

### 3. LES TYPES DE DONNÉES PRÉDÉFINIS

---

```
type élève = {Nom:string; mutable Age:int} ;;
(* Type élève defined. *)
let Martin = {Nom = "Martin"; Age=19} ;;
(* Martin : élève = {Nom = "Martin"; Age = 19} *)
Martin.Age <- Martin.Age + 1; Martin ;;
(* - : élève = {Nom = "Martin"; Age = 20} *)

type point = {mutable X:int; mutable Y:int} ;;
(* Type point defined. *)
let M = {X = 3; Y = 5} ;;
(* M : point = {X = 3; Y = 5} *)
M.X <- (-2); M ;;
(* - : point = {X = -2; Y = 5} *)
let translate m (a,b) = m.X <- m.X+a ; m.Y <- m.Y+b ;;
(* translate : point -> int * int -> unit = <fun> *)
translate M (2,2) ; M ;;
(* - : point = {X = 0; Y = 7} *)
type tortue = {mutable X : float; mutable Y : float; mutable Visée : float} ;;
(* Type tortue defined. *)
```

---

## 4 LES STRUCTURES DE CONTRÔLE

### 4.1 L'expression conditionnelle `if ...then ...else ...`

**Syntaxe** `if Condition then Expression1 else Expression2;;`

- *Condition* doit être une expression *booléenne* (i.e. un prédicat).
- *Expression1* et *Expression2* doivent être du **même type** qui sera alors le type de l'expression `if ...then ...`.

Le seul cas d'« exception » autorisé est celui où l'une des deux expressions est `raise(Exception)` (voir § 6.3).

⚠ **Attention !** Répétons que *Expression1* et *Expression2* doivent être de même type !

⚠ **Attention !** Si *Expression1* ou *Expression2* est une séquence de la forme `action_1; ...; action_p; Expr`, alors elle doit obligatoirement être placée entre `begin` et `end`, sinon le `if ...then ...else` se termine après `action_1`.

⚠ **Attention !** Répétons encore une fois que *Expression1* et *Expression2* doivent être de même type !

```
round x = if x <. 0. then int_of_float(x-.0.5) else int_of_float(x+.0.5) ;;
(* round : float -> int = <fun> *)
```

⚠ **Important !** Il est possible d'utiliser l'instruction `if Condition then Expression1` sans le `else` à condition que *Expression1* soit de type `unit` car `if Condition then Expression1` est équivalent à `if Condition then Expression1 else ();;`.

### 4.2 Les boucles

Il en existe de deux sortes : chacune est une expression de type `unit`.

#### 4.2.1 La boucle inconditionnelle `for`

**Syntaxe** `for compteur = debut to/downto fin do Instructions done;;`

- *compteur*, *debut*, *fin* doivent être des entiers; *compteur* est un identificateur local.
- *compteur* est incrémenté de 1 si `to`, décrémenté si `downto`.
- la boucle n'est pas effectuée si par exemple *fin* < *debut* avec `to`.
- *Instructions* désigne une séquence d'expressions (de type `unit` en général).

#### 4.2.2 La boucle conditionnelle `while`

**Syntaxe** `while Condition do Instructions done;;`

Il s'agit d'une boucle avec test final.

**Remarque** Il est facile de réaliser une boucle avec test initial pour simuler la boucle avec test final : répéter *Instructions* jusqu'à *Condition*. Ainsi :

```
let test = ref true in
while test do
  Instructions;
  if Condition then test := false
done;;
```

### 4.3 Les gardes

On peut combiner filtrage et tests à l'aide du mot-clef `when`.

**Syntaxe**

```
match Expression with
| motif1 when condition1 -> Expr1
| motif2 when condition2 -> Expr2
| .....
| motifN when conditionN -> ExprN ;;
```

## 4. LES STRUCTURES DE CONTRÔLE

---

### Règles

- Les règles du § 2.3 s'appliquent toujours.
- Dans la clause « | *motif* **when** *condition* -> ... », *motif* filtre les mêmes valeurs que précédemment, mais n'est sélectionné que dans le cas où *condition* est vraie. Dans le cas contraire, le filtrage continue normalement en séquence.

```
match x with
| 0. -> 0.
| x when x > 0. -> x *. log x
| _ -> x *. log (-. x);;
```

---

## 5 LE GRAPHISME AVEC CAML

On dispose d'une bibliothèque graphique dont la plupart des fonctions réalisent des « effets de bord » en ce sens qu'elles modifient l'environnement, mais ne renvoient pas de valeurs au sens habituel (renvoient `()`).

### 5.1 Généralités

**Appel à la librairie** Il s'effectue à l'aide de l'instruction `#open "graphics";;` (*ne pas oublier le #*).

#### Ouverture de la fenêtre graphique

Elle se produit avec `open_graph ""` où la fonction `open_graph` est de type `string -> unit`.

La chaîne de caractères suit les conventions de description d'une fenêtre de X-Windows : `"D LxHsXsY"`, où

- `D` désigne le dispositif physique d'affichage (*display*) (s'il est omis, on prend le dispositif par défaut, l'écran de la machine).
- `" LxHsXsY"` désigne la géométrie de la fenêtre proprement dite.
  - `L` et `H` sont deux entiers mesurant la largeur et la hauteur de la fenêtre en pixels.
  - `x` est un séparateur obligatoire entre ces deux nombres.
  - `X` et `Y` sont deux entiers indiquant la position de la fenêtre dans l'écran par le déplacement à effectuer par rapport aux bords.
  - `s` est un signe qui vaut `+` ou `-` :
    - s'il vaut `+` : le déplacement suivant est à compter par rapport au bord gauche de l'écran (pour les abscisses) ou au haut de l'écran (pour les ordonnées).
    - s'il vaut `-` : le déplacement suivant est à compter par rapport au bord droit de l'écran (pour les abscisses) ou au bas (pour les ordonnées).

#### Exemple

```
#open "graphics";;
open_graph " 640x480+100-0";;
```

ouvre une fenêtre graphique à l'écran, d'une taille de 640 par 480, et positionnée en bas et presque complètement à gauche de l'écran.

#### Utilisation de la fenêtre graphique

Le système de coordonnées est le même qu'en maths : l'origine est en bas et à gauche de l'écran et *Ox* est horizontal.

Les fonctions de tracés utilisent comme arguments des *valeurs entières*, en particulier les angles sont exprimés *en degrés* dans le sens trigonométrique.

Les tracés en dehors de la fenêtre ne provoquent pas d'erreurs.

`clear_graph ();;` (de type `unit -> unit`) efface la fenêtre graphique.

`size_x ()` et `size_y ()` (de type `unit -> int`) renvoient la taille de la fenêtre graphique (fixée d'avance).



**Attention !** Les coordonnées des points de l'écran sont **entières** et vont de 0 à `size_x()-1` en *x* et de 0 à `size_y()-1` en *y*.

#### Fermeture de la fenêtre graphique

Une fois terminé, ne pas oublier de fermer la fenêtre graphique avec `close_graph ();;` (de type `unit -> unit`).

### 5.2 Couleurs

`color == int` ce qui signifie que le type `color` est un autre nom pour le type `int`.

Une couleur est caractérisée par ses trois composantes (R,G,B) (Red, Green, Blue).

Chaque composante est un entier pris dans l'intervalle 0..255.

Les trois composantes sont groupées en un entier qui s'écrit : `00xRRVVBB`, où `RR` sont les deux chiffres hexadécimaux pour la composante rouge, `VV` pour la composante vert et `BB` pour la composante bleu.

On dispose pour ce codage de la fonction de conversion suivante :

- `rgb int -> int -> int -> int`

`rgb r v b` renvoie l'entier qui code la couleur associée au triplet `(r,v,b)`.

## 5. LE GRAPHISME AVEC CAML

- `set_color color -> unit`

Fixe la couleur courante du tracé. On dispose des couleurs prédéfinies suivantes :

black, white, red, green, blue, yellow, cyan, magenta

### 5.3 Tracés de points et de lignes

- `plot int -> int -> unit`

Trace le point donné avec la couleur courante du tracé.

- `point_color int -> int -> color`

Renvoie la couleur du point donné. Résultat non spécifié si le point est en dehors de l'écran.

- `current_point unit -> int * int`

renvoie la position du point courant.

- `moveto int -> int -> unit`

Positionne le point courant.

- `lineto int -> int -> unit`

`lineto x y;;` trace une ligne entre le point courant et le point (x,y) et déplace le point courant au point (x,y).

- `draw_arc int -> int -> int -> int -> int -> int -> unit`

`draw_arc x y rx ry a1 a2` trace un arc d'ellipse avec pour centre (x,y), rayon horizontal rx et rayon vertical ry, de l'angle a1 à l'angle a2 (en degrés). Le point courant n'est pas modifié.

- `draw_ellipse int -> int -> int -> int -> unit`

`draw_ellipse x y rx ry` trace une ellipse avec pour centre (x,y), de rayon horizontal rx et de rayon vertical ry. Le point courant n'est pas modifié.

- `draw_circle int -> int -> unit`

`draw_circle x y r` trace le cercle avec pour centre (x,y) et rayon r. Le point courant n'est pas modifié.

- `set_line_width int -> unit`

Fixe la largeur des points et des lignes tracés avec les fonctions ci-dessus.

### 5.4 L'exemple complet de la cardioïde

```
#open "graphics";;
(* fenetre graphique utilisateur *)
let xmin = -8.0 and xmax = 8.0 and ymin = -6.0 and ymax = 6.0;;
open_graph "";
let kx = float_of_int (size_x()) /. (xmax -. xmin);;
let ky = float_of_int (size_y()) /. (ymax -. ymin);;
(* conversion en coordonnees ecran (entieres) *)
let convert x y =
  let round t = if t >= 0. then int_of_float (t +. 0.5)
                else -int_of_float(-. t +. 0.5)
  in (round(kx *. (x -. xmin)), round(ky *. (y -. ymin)));;
let conv f = function (x,y) -> let (u,v)=convert x y in f u v;;
(* Trace d'une courbe parametree *)
let courbe x y tmin tmax nb =
  conv moveto (x(tmin),y(tmin));
  let pas = (tmax -. tmin) /. float_of_int(nb-1) in
  for i = 1 to nb - 1 do
    let t = tmin +. float_of_int(i) *. pas in conv lineto (x(t),y(t)) done;;
(* essai avec une cardioïde *)
set_color red;;
courbe (function t -> cos (2. *. t) +. 2. *. cos (t))
      (function t -> sin (2. *. t) +. 2. *. sin (t))
      (-. 3.14) 3.14 500;
let k = read_key() in close_graph();;
```

### 5.5 Remplissage de figures

- `fill_rect int -> int -> int -> int -> unit`

`fill_rect x y w h` remplit avec la couleur courante, le rectangle dont le coin inférieur gauche est en (x,y) et dont la largeur et la hauteur sont respectivement w et h.



- `fill_poly (int * int) vect -> unit`  
Remplit le polygone donné avec la couleur courante. Le tableau contient les coordonnées des sommets du polygone.
- `fill_arc int -> int -> int -> int -> int -> int -> unit`  
Remplit un secteur d'ellipse avec la couleur courante (mêmes arguments que pour `draw_arc`).
- `fill_ellipse int -> int -> int -> int -> unit`  
Remplit une ellipse avec la couleur courante (mêmes arguments que pour `draw_ellipse`).
- `fill_circle int -> int -> int -> unit`  
Remplit un cercle avec la couleur courante. (mêmes arguments que pour `draw_circle`).

## 5.6 Images

**Le type image** Le type `image` est un type abstrait pour la représentation interne des images. De façon externe, elles sont représentées par des matrices de couleurs (type `color vect vect`).

**La couleur transp** Dans les matrices de couleurs, cette couleur représente un point « transparent » : lorsqu'on trace une image, tous les points de l'écran correspondant à un point transparent dans l'image ne seront pas modifiés, alors que les autres points seront mis à la couleur du point correspondant dans l'image. Ceci permet de surimposer une image sur un fond existant.

### Fonctions

- `make_image color vect vect -> image`  
Convertit la matrice de couleurs donnée en une image. Chaque sous-tableau représente une ligne horizontale. Tous les sous-tableaux doivent avoir la même longueur; sinon, l'exception `Graphic_failure` est déclenchée.
- `dump_image image -> color vect vect`  
Convertit une image en matrice de couleurs.
- `draw_image image -> int -> int -> unit`  
Trace l'image donnée en positionnant son coin inférieur droit au point donné.
- `get_image int -> int -> int -> int -> image`  
Capture le contenu d'un rectangle de l'écran en une image. Les paramètres sont les mêmes que pour `fill_rect`.
- `create_image int -> int -> image`  
`create_image l h` renvoie une nouvelle image de `l` points de large et de `h` points de haut, prête à être utilisée par `blit_image`. Le contenu initial de l'image est quelconque.
- `blit_image image -> int -> int -> unit`  
`blit_image img x y` copie des points de l'écran dans l'image `img`, en modifiant physiquement `img`. Les points copiés sont ceux à l'intérieur du rectangle de coin inférieur droit (`x,y`), et de largeur et hauteur égaux à ceux de l'image.

## 5.7 Tracé de texte

- `draw_char char -> unit` et `draw_string string -> unit`  
Trace un caractère ou une chaîne avec le coin inférieur gauche à la position courante. Après le tracé, la position courante est fixée au coin inférieur droit du texte tracé.
- `set_font string -> unit` et `set_text_size int -> unit`  
Fixe la police et la taille des caractères utilisés pour tracer le texte.
- `text_size string -> int * int`  
Renvoie les dimensions qu'aurait le texte s'il était tracé avec la police et la taille courante.

## 5.8 Sonder la souris et le clavier

On dispose du type énuméré `status` pour rendre compte de l'état de la souris et du clavier, et du type produit `event` pour rendre compte des événements :

```

type status = {
  button : bool;      (* vrai si un bouton de souris enfoncé *)
  mouse_x : int;      (* coordonnée x de la souris *)
  mouse_y : int;
  keypressed : bool;  (* vrai si une touche a été enfoncée *)
  key : char;         (* le caractère de la touche enfoncée *)
};;

type event =
  Button_down (* un bouton de souris enfoncé *)
| Button_up
| Key_pressed (* une touche est enfoncée *)
| Mouse_motion (* la souris est déplacée *)
| Poll        (* ne pas attendre, retourner *)

```

## 5. LE GRAPHISME AVEC CAML

### Primitives usuelles

- `wait_next_event` `event list -> status`

Si Poll (*qui signifie sonder*) est dans la liste, renvoie sans attente le statut courant de la souris et du clavier. Sinon attend l'avènement de l'un des événements spécifié dans la liste d'événements donnée en argument et renvoie alors le statut de la souris ou du clavier à ce moment là.

- `read_key` `unit -> char`

Attend qu'une touche soit enfoncée et renvoie le caractère correspondant. Les caractères tapés d'avance sont conservés dans un tampon et donc pris en compte.

- `key_pressed` `unit -> bool`

Renvoie `true` si un caractère est disponible dans le tampon d'entrée de `read_key`, `false` sinon. Si `key_pressed` vaut `true`, alors le prochain appel à `read_key` renverra un caractère sans attente.

- `button_down` `unit -> bool`

Renvoie `true` si un bouton de la souris est enfoncé, `false` sinon.

- `mouse_pos` `unit -> int*int`

Renvoie le couple des coordonnées de la position du curseur de la souris relativement à la fenêtre graphique. Si le curseur de la souris est hors de la fenêtre graphique, les champs `mouse_x` et `mouse_y` de l'événement sont en dehors du rectangle  $[0..size\_x()-1] \times [0..size\_y()-1]$ . Les caractères tapés sont mémorisés dans un tampon et restitués un par un quand l'événement `Key_pressed` est spécifié.

Exemple d'utilisation : On veut déclencher la fermeture de la fenêtre graphique, après avoir admiré un dessin, en appuyant sur une touche ou sur un bouton de la souris. On dispose d'au moins deux méthodes :

```
let k=read_key() in close_graph() ;;
wait_next_event [Button_down; Key_pressed]; close_graph() ;;
```

On veut récupérer les coordonnées du point où l'utilisateur clique avec la souris :

```
let a = wait_next_event [Button_down] in
  a.mouse_x, a.mouse_y ;;
```

## 5.9 Émettre un son

- `sound` `int -> int -> unit`

`sound freq dur` émet un son de `freq` hertz pendant `dur` millisecondes.

## 5.10 Sauver l'écran graphique

**Utilisation du format Caml** Il faut créer une image en Caml et la sauver sur disque avec `output_value`. Par exemple :

```
(* Écrit une image dans le fichier "file_name" *)
let output_image im file_name =
  let oc = open_out_bin file_name in
  let imv = dump_image im in
  output_value oc imv;
  close_out oc;;

(* Lit une image dans le fichier "file_name" *)
let input_image file_name =
  let ic = open_in_bin file_name in
  let im = make_image (input_value ic) in
  close_in ic;
  im;;

(* Sauve le contenu actuel de la fenêtre graphique. *)
let save_screen file_name =
  let im = get_image 0 0 (size_x ()) (size_y ()) in
  output_image im file_name;;

(* Récupère le contenu de la fenêtre graphique. *)
let restore_screen file_name =
  let im = input_image file_name in
  draw_image im 0 0;;
```

### Sauvegarde d'image au format bitmap

Une méthode est décrite dans « la lettre de Caml » numéro 1, disponible en ligne.

## 6 DIVERS OUTILS CAML

### 6.1 Mesurer le temps en Caml

On utilise la fonction `time : unit -> float` du module `sys`, qui donne en secondes le temps écoulé depuis le lancement du système interactif : On utilisera donc `sys__time ();;` comme dans l'exemple suivant :

```
let chrono = sys__time () in
  (* instructions ici *)
  sys__time () -. chrono ;;
```

### 6.2 Le mode trace

Il est possible de faire afficher les arguments que reçoit une fonction et le résultat renvoyé par celle-ci grâce à l'instruction

```
trace "Nom_Fct";;
```

C'est surtout intéressant pour la mise au point de certaines fonctions ou pour comprendre la récursivité. Cependant, si on veut vérifier comment s'effectue l'évaluation des arguments de cette fonction, il faut aussi déclarer en mode trace les fonctions qui interviennent dans cette évaluation.

Par exemple, dans le cas de la fonction factorielle récursive, si on souhaite voir afficher quels sont les produits d'entiers qui sont effectués, au lieu d'écrire `n*fact(n-1)`, on utilisera `mult_int n fact(n-1)`, puis avant exécution, on déclarera en plus `trace "mult_int";;`

On annule le mode trace d'une fonction avec `untrace "Nom_Fct";;`

### 6.3 Les exceptions

On trouve dans la documentation en ligne la description du type `exn`, ainsi que les fonctions associées. Le mécanisme de traitement des exceptions est expliqué dans la faq en ligne.

### 6.4 Générateur de nombres pseudo-aléatoires

Les fonctions suivantes sont définies dans le module `"random"` (qu'on pourra charger avant appel avec `#open "random"`, sinon mettre le préfixe `random__` devant le nom de chaque fonction du module) :

- `init : int -> unit`

Initialise le générateur aléatoire en utilisant pour germe (*seed* en anglais) l'argument fourni. Le même germe produit toujours la même suite de nombres, ce qui est utile lorsqu'on veut réexécuter un programme lors de sa mise au point avec le même jeu de valeurs aléatoires.

- `int max : int -> int`

Renvoie un entier aléatoire  $n$  avec  $0 \leq n < \text{max}$  et  $\text{max} < 2^{30}$ .

- `float max : float -> float`

Renvoie un nombre flottant aléatoire  $x$  avec  $0 \leq x < \text{max}$ .

```
(* remplissage d'un tableau d'entiers aléatoires *)
let initialise n max = let u = make_vect n 0 in
  for i = 0 to n-1 do u.(i) <- random__int(max); done; u;
(* initialise : int -> int -> int vect = <fun> *)
initialise 5 10 ;;
(* - : int vect = [|0; 9; 6; 2; 2|] *)
```

### 6.5 Les entrées & sorties

Il s'agit de voir comment on peut échanger des informations avec l'extérieur.

#### 6.5.1 Notion de canal

On définit d'abord la notion très générale de **fichier** qui désigne ici aussi bien :

- le clavier
- l'écran
- un fichier au sens courant du mot stocké sur un disque

Les accès aux fichiers sont classés en deux catégories :

- en mode **entrée** ou **lecture** Ex. le clavier.
- en mode **sortie** ou **écriture** Ex. l'écran.

## 6. DIVERS OUTILS CAML

Caml utilise le nom de **canal** (**channel**) plutôt que de fichier.

On dispose de deux types abstraits prédéfinis :

<code>in_channel</code>	canal ouvert en lecture	<code>out_channel</code>	canal ouvert en écriture
-------------------------	-------------------------	--------------------------	--------------------------

et de trois canaux standards prédéfinis :

<code>std_in</code> ou <code>stdin</code>	de type <code>in_channel</code>	relié au clavier
<code>std_out</code> ou <code>stdout</code>	de type <code>out_channel</code>	relié à l'écran
<code>std_err</code> ou <code>stderr</code>	de type <code>out_channel</code>	relié à l'écran

Dans ce qui suit, nous n'évoquerons que les fichiers-texte qu'on peut schématiser par un tableau (illimité théoriquement) dont les éléments sont de type `char`.

### 6.5.2 Ouverture et fermeture de canaux

Pour pouvoir lire (resp. écrire) dans un canal, il faut au préalable le déclarer **ouvert**. Une fois terminé, il ne faut pas oublier de le **fermer**.

#### Primitives utiles

- `open_in` `string -> in_channel`

Ouverture d'un fichier en lecture (*entrée*). Cette fonction prend comme argument une chaîne constituée du nom (avec éventuellement un chemin) du fichier à lire et renvoie un objet abstrait sous forme d'un canal via lequel se feront toutes les opérations de lecture sur ce fichier. Initialement, la « tête de lecture » est positionnée au début du fichier.

- `close_in` `in_channel -> unit`

Fermeture d'un canal ouvert en lecture.

- `open_out` `string -> out_channel`

Ouverture d'un fichier en écriture (*sortie*).

⚠ **Attention !** Si le fichier existe déjà, alors il est détruit; sinon il est créé. La tête d'écriture est initialisée à la position 0.

- `close_out` `out_channel -> unit`

Fermeture d'un canal ouvert en écriture. Avant la fermeture, les caractères en instance dans le tampon associé sont écrits sur le fichier.

### 6.5.3 Fonctions générales d'écriture

- `flush` `out_channel -> unit`

Vide le tampon associé au canal de sortie donné, en achevant toutes les écritures en instance sur ce canal.

- `output_char` `out_channel -> char -> unit`

Écrit un caractère sur le canal de sortie donné.

- `output_string` `out_channel -> string -> unit`

Écrit une chaîne sur le canal de sortie donné.

- `output` `out_channel -> string -> int -> int -> unit`

`output canal tampon index long` écrit `long` caractères de la chaîne `tampon`, à partir de la position `index`, sur le canal de sortie `canal`.

- `output_byte` `out_channel -> int -> unit`

Écrit un entier `n` sous la forme du caractère de code ASCII `n` (modulo 256) sur le canal de sortie donné.

- `seek_out` `out_channel -> int -> unit`

`seek_out canal pos` fixe la position courante d'écriture (*tête d'écriture*) du canal `canal` à `pos`.

- `pos_out` `out_channel -> int`

Renvoie la position d'écriture courante (*tête d'écriture*) du canal donné.

- `out_channel_length` `out_channel -> int`

Renvoie le nombre de caractères total du canal donné. Vider le tampon des caractères en attente avec `flush` avant d'appeler cette fonction.

**Remarque.** Pour écrire un retour à la ligne dans un fichier, on peut écrire le caractère ``\\n`` (ou terminer la chaîne par `\\n`).

### 6.5.4 Fonctions générales de lecture

- `input_char` `in_channel -> char`

Lit un caractère sur un canal d'entrée. Déclenche l'exception `End_of_file` si la fin du fichier est atteinte.

- `input_line` `in_channel -> string`  
Lit des caractères sur le canal d'entrée donné, jusqu'à rencontrer un caractère saut de ligne. Renvoie la chaîne de tous les caractères lus, sans le caractère saut de ligne à la fin.
- `input` `in_channel -> string -> int -> int -> int`  
`input canal tampon index long` tente de lire `long` caractères sur le canal `canal`, en les stockant dans la chaîne `tampon`, à partir du caractère numéro `index`. La fonction renvoie le nombre de caractères lus, entre 0 et `long` (compris). Le résultat 0 signifie que la fin du fichier a été atteinte. Un résultat entre 0 et `long` (non compris) signifie qu'il n'y a plus de caractères disponibles actuellement; `input` doit être appelé à nouveau pour lire les caractères suivants, si nécessaire.
- `input_byte` `in_channel -> int`  
Analogue à `input_char`, mais renvoie le code ASCII du caractère lu.
- `seek_in` `in_channel -> int -> unit`  
`seek_in canal pos` fixe la position de lecture courante du canal `canal` à `pos`.
- `pos_in` `in_channel -> int`  
Renvoie la position de lecture courante (*tête de lecture*) du canal donné.
- `in_channel_length` `in_channel -> int`  
Renvoie la taille totale (nombre de caractères) du canal donné.

### 6.5.5 Fonctions d'affichage à l'écran

Ce sont des fonctions qui écrivent sur le canal standard `std_out` et qui possèdent des abréviations. Par exemple, `output_char std_out` est simplifié en `print_char`.

- `print_char` `char -> unit`  
Affiche un caractère à l'écran.
- `print_int` `int -> unit`  
Affiche un entier.
- `print_float` `float -> unit`  
Affiche un flottant.
- `print_string` `string -> unit`  
Affiche une chaîne de caractères.
- `print_endline` `string -> unit`  
Affiche une chaîne de caractères suivie d'un saut de ligne.
- `print_newline` `unit -> unit`  
Affiche un caractère de saut de ligne et vide la ligne suivante.

### 6.5.6 Fonctions de lecture du clavier

Ce sont des fonctions qui lisent sur le canal standard `std_in` et qui possèdent des abréviations. Par exemple, `input_line std_in` est simplifié en `read_line()`.

- `read_line` `unit -> string`  
Lit des caractères sur le clavier jusqu'à rencontrer un caractère saut de ligne. Renvoie la chaîne de tous les caractères lus, sans le caractère saut de ligne final.
- `read_int` `unit -> int`  
Lit une ligne sur le clavier et la convertit en un entier.
- `read_float` `unit -> float`  
Lit une ligne sur le clavier et la convertit en un flottant. Le résultat n'est pas spécifié si la ligne lue n'est pas la représentation d'un flottant.

### 6.5.7 Un exemple complet

On se propose de créer le fichier *essai.txt*, d'écrire les deux lignes (en corrigeant une erreur volontaire) : « *Au revoir.* » ; « *A bientôt!* » et ensuite de lire ce fichier.

```
(* ouverture du fichier *)
let nom_fichier = let msg = print_string
  "Entrez le nom du fichier a ouvrir en ecriture : " in read_line ();;
(* Entrez le nom du fichier a ouvrir en ecriture : essai.txt *)
(* nom_fichier : string = "essai.txt" *)
let canal = open_out nom_fichier;;
(* canal : out_channel = <abstr> *)
```

## 6. DIVERS OUTILS CAML

```

(* Ecriture dans le fichier *)
let ligne1 = "Au revoir.\n";;
(* ligne1 : string = "Au revoir.\n" *)
let ligne2 = "À bientôt !";;
(* ligne2 : string = "A bientôt !" *)
output_string canal ligne1;;
(* - : unit = () *)
output_string canal ligne2;;
(* - : unit = () *)
out_channel_length canal;;
(* - : int = 0          Ah bon ? *)
flush canal;;
(* - : unit = ()       Vidons le tampon *)
out_channel_length canal;;
(* - : int = 22        Normal ! *)
seek_out canal 5;;
(* - : unit = () *)
output_char canal `v`;;
(* - : unit = ()       Correction du `w` en `v` *)
pos_out canal;;
(* - : int = 6         La tête d'écriture a avancé *)
close_out canal;;
(* - : unit = ()       Fermeture du canal *)
(* Ouverture en lecture du fichier précédent *)
let fichier = open_in "essai.txt";;
(* fichier : in_channel = <abstr> *)
let phrase1 = input_line fichier;;
(* phrase1 : string = "Au revoir." *)
pos_in fichier;;
(* - : int = 11 *)
in_channel_length fichier;;
(* - : int = 22 *)
input_line fichier;;
(* - : string = "À bientôt !" *)
input_line fichier;;
(* Uncaught exception: End_of_file *)
close_in fichier;;
(* - : unit = () *)

```

### 6.6 Faire des calculs en précision arbitraire avec Caml

Caml dispose d'une bibliothèque de calcul en grande précision, sur les nombres rationnels. Pour pouvoir l'utiliser, il faut lancer un système interactif comportant les grands nombres à l'aide de la commande `camllight -lang fr camlnum` dans le *minibuffer* d'emacs, au lancement de Caml. Il est à noter que l'on ne peut à ma connaissance<sup>3</sup> pas utiliser simultanément les bibliothèques `graphics` et `num`.

Les opérations sur les grands nombres sont suffixées par le caractère `/` : par exemple l'addition se note `+/`. On crée des grands nombres par conversion à partir d'entiers ou de chaînes de caractères. On trouvera plus de détails et la présentation des fonctions associées à l'adresse <http://caml.inria.fr/pub/distrib/caml-light-0.74/cl74refman.pdf>.

```

#open "num";;
num_of_string "2/3";;
(* - : num = 2/3 *)
num_of_int 34;;
(* - : num = 34 *)
let n = num_of_string "1/3" +/ num_of_string "2/3";;
(* n : num = 1 *)
let rec fact n =
  if n <= 0 then
    (num_of_int 1)
  else
    num_of_int n */ (fact (n - 1))

```

3. Merci de me contacter si vous y arrivez ([ghaberer@lamartin.fr](mailto:ghaberer@lamartin.fr))!

```
;;  
(* fact : int -> num = <fun> *)  
fact 100;;  
(* - : num = 93326215443944152681699238856266700490715968264381621468592963895217599993229915608941463976156518286 *)  
(*      2536979208272237582511852109168640000000000000000000000 *)
```

---

## 6.7 Conseils de présentation des programmes

---

On trouvera dans la documentation en ligne des conseils sur la présentation des programmes.

**Pseudo loi des espaces** N'hésitez pas à séparer les mots de vos programmes à l'aide de blancs ; la touche d'espacement est la plus facile à trouver, il n'y a pas de raison de s'en priver!

**Pseudo loi de Landin** Traitez l'indentation de vos programmes comme si elle déterminait la signification de vos programmes.





## 7 AIDE-MÉMOIRE RÉSUMÉ D'EMACS

### 7.1 L'indispensable

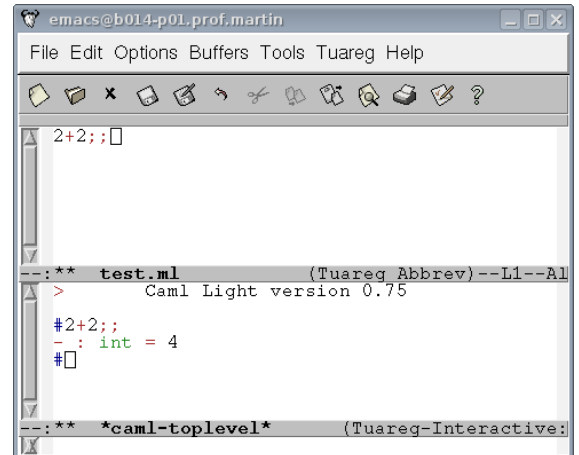
On commence par lancer un terminal dans lequel on tape les instructions :

```
cd /home/repertoire_local
emacs toto.ml &
```

Dans l'utilisation que nous faisons d'Emacs, la fenêtre se décompose en :

- La barre des menus **File | Edit | Options ...**
- Les boutons rapides
- Le buffer contenant le fichier `toto.ml` à éditer
- La ligne d'état du buffer `toto.ml`
- Le buffer `*caml-toplevel*` où Caml est exécuté interactivement
- La ligne d'état du buffer `*caml-toplevel*`
- La **ligne de dialogue** (qu'il faut regarder!)

On peut se déplacer en utilisant la souris, utiliser les menus déroulant, voire les menus contextuels, mais il est préférable d'utiliser les raccourcis suivants :



Commande	Description
C-c C-e	Envoyer l'instruction courante (contenant le curseur) à l'interpréteur Caml <span style="float: right;">tuareg</span>
C-c C-b	Envoyer tout le fichier (en fait le buffer) à l'interpréteur Caml <span style="float: right;">tuareg</span>
C-x C-s	Enregistrer le fichier du buffer courant (contenant le curseur)
C-x C-c	Quitter Xemacs
C-g	Arrêter tout ce qui est en cours.
C-x C-u ou C-_	Annuler la dernière frappe
C-w	Couper le texte sélectionné

### 7.2 Mieux comprendre le terminal

Comprenons les instructions

```
cd /home/repertoire_local
emacs toto.ml &
```

- La première commande a pour effet de nous placer dans le `repertoire_local`, lui-même situé dans `home`, lui-même situé à la racine de l'arborescence.
- L'esperluette qui termine la seconde instruction permet de lancer emacs en tâche de fond : on peut lancer d'autres instructions depuis le terminal.
- Si la fenêtre du terminal est fermée, les commandes lancées depuis cette fenêtre sont terminées. Tant qu'emacs est utilisé, il faut donc laisser le terminal ouvert (ou masqué).
- Il est important d'utiliser un fichier portant l'extension `.ml`. C'est en effet cette extension qui permet de charger le mode tuareg, adapté pour Caml.
- On peut utiliser la « complétion automatique » dans le terminal, grâce à la touche de tabulation, ainsi que d'autres commandes :

Commande	Description	Exemple
ls <-al>	lister les fichiers et sous répertoires du répertoire courant	ls -l
man	afficher la page d'aide (q pour quitter)	man ls
cd ./ ../	changer le répertoire courant le répertoire courant, le répertoire parent	cd ../
a2ps	imprimer un fichier	a2ps toto.ml
cp	copier un fichier	cp toto.ml tata.ml
less	lire un fichier texte	less ./tata.ml
ps aux kill -9 pid	lister les processus tuer le processus <i>pid</i> (Attention!)	ps aux

### 7.3 Aller plus loin avec Emacs et Tuareg

Les commandes Emacs sont constituées d'une combinaison de touches : une touche appelée « modifieur » telle que CTRL ou ESC, suivie d'un ou deux caractères. Par convention :

- **C-g** signifie garder la touche CTRL enfoncée et taper la lettre **g** ;
- **ESC w** signifie taper sur la touche ESC, la relâcher puis taper sur la lettre **w**.

Les commandes spécifiques au mode Tuareg sont indiquées. Les autres sont utilisables pour tout fichier édité avec Emacs.

Commande	Description
<b>C-c C-f</b>	Ouvrir un fichier s'il existe ou créer un nouveau fichier s'il n'existe pas
<b>C-x C-w</b>	Enregistrer sous (permet d'avoir une nouvelle copie du fichier)
<b>C-a</b> / <b>C-e</b>	Aller au début / à la fin de la ligne
<b>C-v</b> / <b>ESC v</b>	Descendre / Monter d'un écran
<b>ESC &lt;</b> / <b>ESC &gt;</b>	Aller au début / à la fin du fichier
<b>ESC g n</b>	Aller à la ligne <i>n</i>
<b>C-espace</b> ou <b>C-@</b>	Marquer le point courant (le déplacement du curseur permet de sélectionner une région)
<b>C-w</b>	Couper la région sélectionnée
<b>C-k</b>	Couper la fin de la ligne
<b>ESC w</b>	Copier la région sélectionnée
<b>C-y</b>	Coller ( <i>yank</i> ) la dernière région coupée ou copiée
Molette souris	Coller la dernière région sélectionnée
<b>C-x h</b>	Sélectionner la totalité du buffer
<b>ESC d</b>	Effacer la fin du mot contenant le curseur
<b>C-x 2</b>	Partager l'écran en 2
<b>C-x o</b>	Aller dans l'autre partie de l'écran
<b>C-x 1</b>	Revenir à un écran non partagé
<b>C-l</b>	Déplacer le texte pour que le curseur soit au milieu de l'affichage
<b>C-s</b>	Recherche incrémentielle avant ( <b>C-s</b> pour occurrence suivante, <b>Enter</b> pour finir)
<b>C-r</b>	Recherche incrémentielle arrière ( <b>C-r</b> pour occurrence précédente, <b>Enter</b> pour finir)
<b>ESC %</b>	Faire un chercher/remplacer ( <b>y</b> remplacer, <b>n</b> ne pas remplacer, <b>!</b> remplacer tout)
<b>C-x C-b</b>	Afficher la liste des buffers
<b>C-x b</b>	Placer le curseur dans un autre buffer
<b>C-x k</b>	Fermer le buffer ( <i>kill</i> )
<b>ESC !</b>	Exécuter une commande shell
<b>TAB</b>	Créer un retrait adapté à l'algorithme
<b>ESC ;</b>	Insérer un commentaire (* *)
<b>C-c C-r</b>	Envoyer la région sélectionnée à l'interpréteur Caml
<b>C-c C-k</b>	Tuer l'interpréteur *caml toplevel*
<b>C-c C-s</b>	Lancer l'interpréteur *caml toplevel*
<b>C-c . i</b>	Insérer <code>if ... then ... else ...</code>
<b>C-c . w</b>	Insérer <code>while ... do ... done</code>
<b>C-c . f</b>	Insérer <code>for ... do ... done</code>
<b>C-c . b</b>	Insérer <code>begin ... end</code>
<b>C-c . m</b>	Insérer <code>match ... with ...</code>
<b>C-c . l</b>	Insérer <code>let ... in ...</code>
<b>C-c . t</b>	Insérer <code>try ... with ...</code>

Plus d'information sur wikipedia ou sur le site de Thomas Vergnaud.

## 8 AIDE-MÉMOIRE RÉSUMÉ DE CAML

### Directives générales.

commentaires  
appel d'un module compilé  
chargement d'un programme source  
déclarations globales de valeurs  
déclarations locales de valeurs  
séquence d'expressions

```
(* ... *)
#open "nom du module";;
include "nom du programme";;
let ident1=valeur1 and ...;;
let ident1=valeur1 and ...in Expression;;
begin action_1; ...; action_p; Expr; end;;
```

### Déclarations de fonctions.

fonction sans argument  
fonction à un argument  
  
fonction à  $n$  arguments curryfiée  
fonction à  $n$  arguments non curryfiée  
fonction avec filtrage

```
let f() = Expression;;
let f(x) = Expression;;
ou let f x = Expression;;
ou let f = function x -> Expression;;
let f x1 x2 ... xn = Expression;;
let f(x1, ... ,xn) = Expression;;
let f = function | motif1 -> Expr1
                  | motif2 -> Expr2 ...;;
ou utiliser match
let rec f ...;;
let prefix nom x y = ...;;
```

### Entiers. (de type int)

opérations arithmétiques  
comparaison  
fonctions prédéfinies  
entier aléatoire entre 0 et  $n - 1$   
conversion entier  $\rightarrow$  chaîne  
conversion chaîne  $\rightarrow$  entier

```
+, -, *, /, mod
=, <, <=, >, >=, <>
abs, succ, pred, max, min
random__int(n);;
string_of_int
int_of_string
```

### Réels. (de type float)

opérations arithmétiques  
comparaison  
  
fonctions prédéfinies  
  
réel aléatoire entre 0 et  $a$   
conversion réel  $\rightarrow$  chaîne  
conversion chaîne  $\rightarrow$  réel  
conversion réel  $\rightarrow$  entier  
conversion entier  $\rightarrow$  réel

```
+. , -. , *. , /. , **
=., <., <=., >., >=., <>.
ou =, <, <=, >, >=, <>
abs_float, sqrt, exp, log (népérien)
sin, cos, tan, asin, acos, atan (radians)
random__float(a);;
string_of_float
float_of_string
int_of_float
float_of_int
```

### Caractères et chaînes de caractères.

syntaxe d'un caractère, d'une chaîne  
longueur de la chaîne  $s$   
concaténation de chaînes  
accès au caractère d'indice  $i$  de  $s$   
modification d'un caractère de  $s$   
création  
extraction d'une sous-chaîne de  $s$   
conversion caractère  $\rightarrow$  chaîne

```
`a` "azerty"
string_length s;;
avec ^
s.[i] (remarquer les crochets)
s.[i] <- valeur (les indices commencent à 0)
make_string longueur caractère_initial
sub_string s début longueur
string_of_char c
```

## 8. AIDE-MÉMOIRE RÉSUMÉ DE CAML

### Booléens. (de type `bool`)

vrai, faux  
non, ou, et  
comparaison

`true, false`  
`not, or (ou ||), & (ou &&, mais pas and)`  
`=, <, <=, >, >=, <>`

### Listes. (de type `'a list`)

entrée ou affichage  
ajout de `x` en tête d'une liste  
liste vide, concaténation  
longueur d'une liste  
test d'appartenance  
position d'un élément  
tête et queue d'une liste  
fonction récursive typique sur une liste

`[x1; x2; ... ;xn]`  
`x::liste`  
`[], @`  
`list_length liste`  
`mem élt liste`  
`index élt liste`  
`hd(liste) (head) et tl(liste) (tail)`  
`let rec f l = match l with`  
    `| [] -> ...`  
    `| t :: q -> ...;;`

### Tableaux ou vecteurs. (de type `'a vect`)

entrée ou affichage  
longueur du tableau `t`  
accès à l'élément d'indice `i` de `t`  
modification d'un élément de `t`  
création d'un vecteur  
copie de `t`  
extraction dans `t`  
création d'une matrice  
accès à `A(i,j)`

`[|t0; t1; ... ;tp|]`  
`vect_length t;;`  
`t.(i)` (remarquer les parenthèses)  
`t.(i) <- valeur` (les indices commencent à 0)  
`make_vect longueur valeur_initiale`  
`copy_vect v`  
`sub_vect v début longueur`  
`make_matrix m n valeur_initiale`  
`A.(i).(j)`

### Références. (de type `'a ref`)

déclaration  
accès à la valeur pointée  
modification

`let nom = ref valeur`  
`!nom`  
`nom := nouvelle_valeur`

### Structures de contrôle.

test  
boucle avec compteur  
boucle `tant que`  
filtrage

`if ... then ... else ...;;`  
`for i = début to/down to fin do ... done;;`  
`while Condition do ... done;;`  
`match Expr with` | motif1 -> Expr1  
    | motif2 -> Expr2 ...;;

### Entrées - sorties.

affichage à l'écran  
retour à la ligne suivante  
lecture au clavier

`print_char, print_int, print_float, print_string`  
`print_newline();;`  
`read_int, read_float, read_line`

### Exceptions.

Déclenchement

`failwith "Message";;`