Les bases de Caml

S'initier au langage de programmation Caml, et découvrir les principaux aspects de sa syntaxe au travers d'exemples simples : tel est l'objectif de cette première leçon.

Vous pouvez consulter l'aide en ligne pour approfondir certains aspects de ce langage.

Par ailleurs, la lecture du présent texte doit être accompagnée de l'expérimentation sur machine : ce n'est qu'en essayant les exemples proposés, en explorant des variantes de ceux-ci et en observant les réponses du système (et notamment les messages d'erreurs ...) qu'il sera possible d'assimiler les rudiments mais aussi les subtilités du langage.

I. Brève présentation de Caml

Caml est un langage destiné à l'apprentissage de certains aspects fondamentaux de l'algorithmique et de la programmation, et en aucun cas au développement d'applications professionnelles.

Trois fenêtres sont généralement regroupées dans une même interface :

– Entrée (où l'on tape des phrases en Caml) ;
– Sortie (où les résultats des évaluations apparaissent) ;
– Historique (comme son nom l'indique ...).

Une quatrième fenêtre, dédiée au graphisme, peut éventuellement s'ajouter aux précédentes.

L'implémentation Caml light que nous utiliserons se présente comme une boucle interactive :

→ Caml invite l'utilisateur à écrire une phrase ;
→ l'utilisateur tape une phrase, c'est-à-dire, le plus souvent, une expression ou une déclaration, qu'il termine par « ;; », et valide en appuyant sur entrée ;
→ Caml analyse alors la syntaxe de cette phrase ;
→ lorsque cette syntaxe est correcte, il détermine le type de l'expression ;
→ si cette expression est bien typée, il l'évalue, puis renvoie une réponse sous la forme :

– : type = valeur.

Remarque : à chacune des étapes évoquées, Caml peut renvoyer un message d'erreur.

Remarque : cette manière de programmer, au sein de cette boucle interactive, est conviviale. Elle permet d'utiliser Caml comme une calculatrice, ou un peu comme Maple (en moins souple). Elle a cependant des limites évidentes, dès que le programme écrit est élaboré. On pourra donc, le cas échéant, écrire un fichier en .ml (avec Notepad ++ par exemple) , que l'on ouvrira dans l'implémentation Caml light.

II. Les Expressions.

1. Phrases en CAML.

Une **phrase** ou **requête** peut occuper plusieurs lignes ; elle se termine par un double point-virgule **;;**.

```
1+1;;          → phrase
– : int = 2    → évaluation
```


Le résultat de l'évaluation (quand elle a lieu !) de chaque phrase comporte trois indications :
– le **nom** de la variable globale qui vient d'être définie le cas échéant ou, à défaut, un simple tiret.
– le **type** du résultat.
– la **valeur** du résultat.

En résumé :

Ecriture d'une phrase `phrase ;;`

Evaluation d'une phrase `nom : type = valeur` ou `- : type = valeur`

2. Commentaires.

Il est conseillé de placer des **commentaires** dans le code source ; un commentaire consiste en du texte, ignoré par l'interpréteur. Ce texte est donc exclusivement destiné à l'humain qui le lira. L'emploi de commentaires rend les sources d'un programme lisibles par un tiers (qui, bien souvent, est l'auteur quelques temps plus tard). Un commentaire se doit d'être concis, et ne consiste pas en un "plagiat" du code source : il doit éclairer ce dernier, préciser les sous-entendus ainsi que le principe de la méthode employée,...

Les délimiteurs de commentaires sont (* et *).

En résumé :

Insertion de commentaires `(* ... *)`

III. Les identificateurs

Il est souvent utile de donner un nom au résultat d'un calcul effectué par Caml. Comme en mathématiques, on peut avoir besoin de « variables » globales et locales.

1. Identificateurs globaux

On parle aussi, en se conformant à l'usage, de variables globales, même si nous verrons que cette expression peut induire en erreur. Les identificateurs globaux sont conservés en mémoire, et donc utilisables dans l'intégralité du programme. Ils sont à réserver aux objets dont nous avons besoin tout au long de notre travail.

Pour introduire (ou définir, ou déclarer) un identificateur global, la syntaxe est :

let *identificateur* **=** *valeur*

Une telle définition est une liaison d'un identificateur (autrement dit d'un nom, d'une abréviation) à une valeur (qui devient ainsi identifiée), que l'on peut utiliser dans la suite de notre programme. C'est l'équivalent du « soit » en mathématiques, tel qu'employé dans l'expression « Soit M la borne supérieure de f sur [0; 1]. »

```
let y = 7 ;; (* 7 a pour nom y *)
y : int = 7
let successeur n = n + 1 ;;
successeur : int -> int = <fun>
successeur y ; ; (* Caml sait qui est désigné par le symbole y *)
– : int = 8
let y = 7 ;;
y : int = 7
y + 2 ;;
– : int = 9
```

Remarque : voici un point relativement compliqué. Il ne faut pas confondre cette définition de l'affectation. Une fois défini, un nom conserve la valeur donnée, à moins de le redéfinir, c'est pourquoi le terme de « variable » est plutôt à éviter dans ce cadre, et il vaut mieux parler d'identificateur.

```
let y = 7 ;; (* 7 a pour nom y *)
y : int = 7
let x = y + 2 ;; (* 7 + 2 a pour nom x *)
x : int = 9
let y = 2 ;; (* y n' identifie plus 7 mais 2 *)
y : int = 2
x ;; (* x est toujours le nom de 9 *)
- :  int = 9
```

Pour une raison obscure, nous avons redéfini *y*. Nous pouvons même le redéfinir à partir de sa définition actuelle :

```
let y = 7 ;; (* 7 a pour nom y *)
y : int = 7
let y = y + 2 ;; (* 7+2 a pour nom y *)
y : int = 9
```

Cependant, cette programmation horrible est à proscrire, tant elle contrevient à l'esprit Caml. D'une manière générale, il ne semble pas opportun de redéfinir un identificateur.

2. Identificateurs locaux

Ce sont les identificateurs auxiliaires, permettant de réaliser et de donner un sens à un calcul secondaire : ce sont des définitions « jetables » (et « jetées »).
La syntaxe est la même que pour les identificateurs globaux, mais on lie la déclaration de l'identificateur local à l'expression dans laquelle il est utilisé par **in** :

let *identificateur_local* **=** *valeur* **in** *expression*

```
# let y = 7 in y + 2 ;;
- : int = 9
# y ;;
Toplevel input :
>y ;;
>^
The value identifier y is unbound .
```

On peut aussi observer que cette phrase est une expression et non une déclaration.
On peut par ailleurs utiliser un identificateur local pour définir un identificateur global :

```
let x = (* x est global *) let y = 7 in (* y est local, et sert à définir x *) y + 2 ;;
x : int = 9
x ;;
- : int = 9
y ;;
Toplevel input :
>y ;;
>^
The value identifier y is unbound .
```

En reprenant l'analogie avec l'exemple de la somme en mathématiques, que se passe-t-il si nous avons la mauvaise idée d'utiliser un identificateur global pour identificateur local ?


```
let y = 3 ;;
y : int = 3
let x = let y = 7 in y + 2 ;;
x : int = 9
x ;;
- : int = 9
y ;;
- : int = 3
```

L'identificateur local l'a emporté au sein de l'expression à laquelle il est associé, ce qui est normal (à quoi servirait-il sinon ?). Bien sûr, il vaut mieux éviter cette réutilisation pour le moins maladroite.
Pour mieux insister sur le caractère secondaire d'un identificateur local, on peut inverser les ordres de la définition et de l'expression, en utilisant **where** au lieu de **in**, c'est-à-dire en écrivant :

expression **where** *identificateur_local* **=** *valeur*

On peut ainsi écrire :

```
# let x = y + 2 where y = 7 ;;
x : int = 9
# x ;;
- : int = 9
```

Remarque : dans la mesure du possible, on privilégiera les identificateurs locaux ; cela limitera les erreurs.

Afin d'éviter de trop nombreuses définitions emboitées, on pourra utiliser *and* pour introduire plusieurs identificateurs (qu'ils soient locaux ou globaux, et que ce soit avec *in* ou *where*) :

```
# x + y + z  where z = 13 and x = 5 and y = 12 ; ;
- : int = 30
```

En résumé :

Liaison globale : `let identificateur = valeur`

Liaison locale : `let identificateur = valeur in expression` | `expression where identificateur = valeur`

Liaisons simultanées : `let identificateur_1 = valeur_1 and identificateur_2 = valeur_2 ...`

Exercice 1.

Définir la variable *y* dont la valeur est *x* + 3, *x* étant une variable locale de valeur 4.

Exercice 2.

Typer et évaluer les phrases suivantes d'une même session CAML:

```
→let a = 80;;
→let b = a + 20;;
→let a = b + 50 in a + a * 3 ;;
→let a = 3 in let b = a + 2 in a + b;;
→let a = 3 in let a = 5 and b = a + 2 in a + b;;
→3 + (let a = 2 in 3*a);;
```

Exercice 3.

Calculer $\dfrac{\cos(\ln 3)+\sin(\ln 2)}{\cos^3(\ln 3)-\sin^2(\ln 2)}$.

IV. Types élémentaires

On recense ici les types élémentaires en Caml, i.e. ceux qui ne sont pas construits à partir d'autres types : par exemple, le type *int* (celui des entiers) est élémentaire, pas le type *int –> int* (celui des fonctions qui à un entier associent un entier). Nous avons déjà parlé du type *unit*. Il ne s'agit pas ici de décrire de manière exhaustive ces types et les fonctions associées, mais de présenter les plus utiles d'entre eux.

1. Le type booléen

Le type *booléen* ne prend que deux valeurs : *true* et *false* .
Un **prédicat** est une fonction à valeurs booléennes (c'est surtout à cela que servent les booléens).
On dispose du connecteur logique « et », s'écrivant & ou &&, du connecteur logique « ou », s'écrivant *or* ou || , et du connecteur logique « non » s'écrivant *not*.

Remarque : Ces opérateurs binaires « et » et « ou » sont dits **infixes** car ils se placent entre les arguments.

Remarque : Faites attention, *and* ne désigne pas le connecteur logique « et » !

Remarque : Il faut savoir que Caml effectue une évaluation dite paresseuse, ce qui signifie que dès qu'il dispose d'informations suffisantes pour déterminer la valeur du booléen considéré, il arrête le parcours ; par exemple, l'expression (1>2) && (2<3) est évaluée faux sans que (2<3) ne soit évalué.

A retenir :

→Opérations : =, **||** ou **or** (☞ou), **&** ou **&&** (☞ et), **not** (☞ non).

```
(1<2) && (2012=2013);;
– : bool = true
```

2. Le type int

C'est, comme nous l'avons dit, le type des entiers. Il faut cependant prendre garde, car Caml gère seulement les entiers compris entre -2^{30} et $2^{30}-1$, et se ramène à un tel entier par congruence modulo 2^{31}. On obtient donc des résultats étonnants si on ne fait pas attention :

```
10000 * 10000 * 10000 ;;
– : int = –727379968
```

On dispose des opérateurs binaires (à deux arguments) infixes +, – , *, / et *mod*, dont les définitions sont claires. Précisons simplement que *a* / *b* et *a mod b* désignent respectivement le quotient et le reste de la division euclidienne de *a* par *b* lorsque *a* et *b* sont des entiers naturels (*b* non nul), mais que (–7) *mod 5* par exemple vaut –2 (et non 3).

A retenir :

→Opérations : +, –, *, /(☞ quotient entier), mod (☞ reste de la division euclidienne).
→Comparaisons : < , > , <= , >= , = , <>.
→Fonctions de base : *min*, *max*, *abs*, *random__int*(n) (☞entier aléatoire dans [0; n–1]);
→La fonction *print_int* permet d'afficher un entier.

```
1+1;; - : int = 2
2*3;; - : int = 6
7/3;; - : int = 2
7 mod 3;; - : int = 1
random__int 6;;  - : int = 4
```


3. Le type float

Le type *float* est celui des nombres à virgule flottante (en bref : des flottants), qui sont des approximations des nombres réels.
Remarque : la virgule dont il est question s'écrit avec un point (3.5, 4. sont acceptés, mais pas .6) :

```
2 . 6 ;;
– : float = 2 . 6
```

Les opérateurs sur ces nombres sont pour l'essentiel ceux définis pour les entiers *int*, suivis d'un point, i.e. +., _., *. et /..
On dispose aussi de l'exponentiation ** :

```
2. ** 10. ;;
– : float = 1024.0
```

Retenez bien que ** s'emploie pour les nombres à virgule flottante, et non les entiers.

Remarque : pour de grands nombres, Caml passe en notation scientifique mantisse-exposant.

```
3. ** 45.;;
– : float = 2.95431270655e+21
```

Remarque : On peut utiliser les opérateurs de comparaison =., <., <=., >=. et <> mais en réalité, il n'est pas nécessaire de mettre de point après ces symboles lorsqu'on traite de *float*.

Enfin, les fonctions usuelles *sqrt, exp, log, sin, cos, tan, acos, asin et atan*, dont il est inutile de donner la définition, sont des primitives Caml.

A retenir :

→Opérations : +., –., *., / , ** ou **. (☞ puissance);
→Comparaisons : <. ou <, >. ou >, <=. ou <=, >=. ou >=, =. ou = ,<>. ou <>. ;
→Fonctions de base : *abs_float, sin, cos, asin, acos, atan, exp, log, sqrt, cosh, sinh,*
floor (☞sol) et *ceil* (☞plafond) : arrondissent un nombre réel à l'entier immédiatement inférieur et supérieur mais le résultat étant de type réel,
random__float(a) (☞ réel aléatoire dans [0; a[) ;
→La fonction *print_float* permet d'afficher un flottant.

```
1. +. 2.1;; - : float = 3.1
2. ** 3.;;  - : float = 8.0
10e2. ;;   - : float = 100.0
let pi = atan(1.) *. 4. ;; pi : float = 3.14159265359
```

4. Le type char

C'est le type des caractères. On les écrit entre accents graves (pour ne pas les confondre avec un identificateur).

```
`a`.;;
– : char = `a`
```

A retenir :

→Fonctions de base : *char_of_int* (☞code ⟶ caractère), *int_of_char* (☞caractère ⟶ code) ;
→La fonction *print_char* permet d'afficher un caractère.

```
char_of_int 65 ;;   - : char = `A`
int_of_char `a`. ;; - : int = 97
```

5. Le type string

Une chaîne de caractères est délimitée par des guillemets : " ".

Les termes d'une telle chaîne de caractères *s* sont indicées, le premier indice étant 0 (et non 1).
Pour accéder au terme d'indice *i* de *s*, on écrit simplement *s.*[*i*] .
Pour le modifier, la syntaxe est *s.*[*i*] <– *nouvelle_valeur* où *nouvelle_valeur* est un caractère.

```
let message = “bonjour !” ;;
 message : string= “bonjour !”
message.[0] <– `B ;;
 - : unit = ()
message ;;
- : string = "Bonjour !"
string_length message ;;
 - : int = 9
```

A retenir

→Opérations : =, <>;
→Fonctions de base : ^(☞ concaténation), *string_length* (☞ longueur), *make_string* (☞ création d'une chaîne), *sub_string* (☞sous-chaîne), etc…
→Caractère de rang *i* (☠ numérotation à partir de 0) : S.[*i*] (☞ lecture), S.[*i*] <- caractère (☞ affectation) ;
→La fonction *print_string* permet d'afficher une chaîne de caractères.

Remarque : Les chaînes de caractères sont des types dits **mutables** car elles peuvent être modifiées (sur place) ; elles se comportent comme des vecteurs (cf. leçon suivante).

Type chaîne de caractères `string`

Accès au caractère de rang i : en lecture `chaine.[i]` en écriture `chaine.[i] <- caractère`

6. Conversion de types élémentaires

Caml dispose de nombreuses fonctions de conversion de type.

Leurs noms sont explicites : *int_of_float* , *float_of_int* , *string_of_float* , etc.

Précisons que *type1_of_ type2* convertit un objet de type *type2* en un objet de type *type1*.

7. Affichage de types élémentaires

On utilise les procédures suivantes, aux noms transparents : *print_int* , *print_float* , *print_char*, *print_string* .

A ce propos, on peut produire un retour à la ligne grâce à *print_newline* (), *print_char* `\n` ou *print_string* "\n" au choix.

Exercice 4.

Déterminer le type et la valeur affichés par Caml pour chacune des expressions suivantes :

→ `5 mod 2 ;;` → `1.2 + 3.4 ;;` → `5/2 ;;` → `1.1e-3 +.2. ;;`

V. Deux exemples de types non élémentaires

1. Les n-uplets

Un **multiplet** est un expression dont le type est le produit cartésien de plusieurs types.

Par exemple :

```
"Hugo","Victor";;
- : string * string = "Hugo", "Victor"
1,3.14;;
- : int * float = 1, 3.14
1,2,3,4;;
 - : int * int * int * int = 1, 2, 3, 4
1,(2,3);;
- : int * (int * int) = 1, (2, 3)
```

La virgule ',' est le constructeur de multiplet.

Des parenthèses peuvent être utilisées pour améliorer la lisibilité ou éviter une ambiguïté :

```
(1,2,3,4);; - : int * int * int * int = 1, 2, 3, 4
```

Remarque : Les objets *a* et *b* n'ont pas nécessairement le même type.

En résumé :

Définition d'un multiplet `(valeur_1,valeur_2,…,valeur_n)`

Remarque : On accède à la première (resp. seconde) composante par les fonctions *fst* et *snd*, mais il est facile de redéfinir ces fonctions.

Remarque : pour récupérer une composante quelconque dans un multiplet quelconque :

```
let  (_,_, z,_) = (1,5,4,8) ;;
z : int = 4
```

2. Les fonctions

Nous allons voir à la section suivante comment définir des fonctions. Mentionnons simplement ici que le type *type1* –> *type2* est celui des fonctions dont l'argument est de type *type1*, dont les valeurs sont de type *type2*.

VI. Programmation fonctionnelle

Etant un langage dit **fonctionnel**, les **fonctions** sont donc le concept central de Caml .

Un programme Caml est souvent une suite de **fonctions**, que l'on applique à la fin. Il est donc très important de savoir définir et utiliser les fonctions en Caml. La plupart des fonctions que nous définirons seront récursives, mais on s'intéresse seulement dans cette leçon aux fonctions non récursives.

Une fonction en Caml correspond assez bien à l'idée que l'on se fait d'une fonction en mathématiques.

Elle accepte un (ou plusieurs) **argument**(s) et renvoie un **résultat**.
Les arguments et le résultat appartiennent tous à des types bien déterminés; d'ailleurs, lors de la définition d'une fonction, Caml nous informe du type de celle-ci.

1. Fonctions à un argument.

Déclaration :

```
let double = function x -> 2*x;;
double : int -> int = <fun>
```

ou, syntaxiquement plus léger mais qui revient exactement au même :

```
let double x = 2*x;;
double : int -> int = <fun>
```

Le mot-clef **let** a déjà été rencontré : il introduit la définition de la fonction.

Type :

Le système évalue la phrase et répond que *double* est une fonction qui, à un entier (*x*), associe un entier (2*x).
C'est le sens de la notation qui définit le type de la fonction : *int -> int*.

Appel :

```
double (10 + 1);; - : int = 22
```

L'expression 10+1 est évaluée, puis sa valeur (= 11) est substituée au paramètre formel *x*, qui lui correspond dans la définition de la fonction.

L'emploi des parenthèses est optionnel, à condition de respecter une règle : en Caml, l'application d'une fonction, qui est notée par simple juxtaposition, possède la plus forte priorité :

```
double 11;;
- : int = 22
double (10 + 1);;
 - : int = 22
double 10 + 1;;
 - : int = 21
double -2;;
This expression has type int -> int, but is used with type int.
double (-2);;
 - : int = -4
```

D'une manière générale, l'expression x y z désigne (x y) z. Pour obtenir x (y z), il faut le demander explicitement :

```
let u x = x + 1;;
u : int -> int = <fun>
let v x = x * x;;
v : int -> int = <fun>
u v 3;;
This expression has type int -> int, but is used with type int.
u(v 3);;
- : int = 10
```


Remarques :

- Les parenthèses sont facultatives autour des arguments lors de la définition ou de l'appel;
- Une fonction renvoie toujours un résultat : la dernière valeur calculée ;
- Il est possible de définir localement des variables (et des fonctions) à une fonctions.

Remarque : on tente le plus souvent de se débarrasser du parenthèsage. Pour ce faire, on peut retenir que :

→*f x + g y* équivaut à *f*(*x*) + *g*(*y*)

→*f x y* équivaut à (*f x*) *y* i.e (*f*(*x*))(*y*)

En résumé :

Déclaration d'une fonction `let nom_fonction argument = expression`

ou `let nom_fonction = function argument -> expression`

Appel d'une fonction à un argument `nom_fonction argument`

Type d'une fonction à un argument `type_argument ->type_résultat`

Remarque : Une fonction peut être définie localement à une autre fonction.

```
let quadruple x =
 let double x = 2*x in
 double (double x);;
```

Exercice 5.

Définir une fonction *f* qui prend en paramètre un entier *x* et retourne en résultat x+2.
Y a-t-il plusieurs façons (syntaxiques) de définir *f* ?
Y a-t-il plusieurs façons (syntaxiques) d'appliquer *f* à l'entier 5 ?
Comment devez-vous faire pour appliquer *f* à 3 * 7 ?

Exercice 6.

Ecrire une fonction nieme_lettre qui étant donné un entier *n* (entre 1 et 26) donne la nième lettre de l'alphabet.

Exercice 7. *Fonction th*

Définir la fonction th (tangente hyperbolique) définie par :

$$th\ x = (e^x - e^{-x})/(e^x + e^{-x})$$

en ne faisant qu'un calcul d'exponentielle.

2. Fonctions à plusieurs arguments.

Déclaration :

```
let produit = fun x y -> x*y;;
produit : int -> int -> int = <fun>
```

ou, syntaxiquement plus léger mais qui revient exactement au même :

```
let produit x y = x*y;;
produit : int -> int -> int = <fun>
```

Type :

Le système répond que produit est une fonction qui, à un entier (x), associe une fonction (x → (y → x*y)) qui à un entier (y) associe un entier (x*y). C'est le sens de la notation *int -> int -> int*.

Autrement-dit, la fonction *produit* aurait pu être définie ainsi :

```
let produit = function x -> function  y -> x*y;;
produit : int -> int -> int = <fun>
```

Appel :

```
produit 3 4;;
- : int = 12
(produit 3) 4;;
- : int = 12
```

Remarque : Une fonction de 2 arguments par exemple peut donc s'interpréter de 2 façons :

→ comme une fonction à 2 arguments !

→ comme une fonction à 1 argument (le 1er) qui renvoie une fonction à 1 argument (le 2ième) .

En résumé :

Déclaration d'une fonction à plusieurs arguments `let nom_fonction arg_1 arg_2 … = expression`

ou `let nom_fonction = fun arg_1 arg_2 … -> expression`

ou `let nom_fonction = function argt_1 -> function arg_2 -> … -> expression`

Appel d'une fonction à plusieurs arguments `nom_fonction argument_1 argument_2 …`

Type d'une fonction à plusieurs arguments `type_argument_1 ->type_argument_2 ->… ->type_résultat`

Exercice 8.

Ecrire une fonction nb_racines qui prend en entrée 3 arguments $a \neq 0$, b et c et renvoie le nombre de solutions dans $\mathbb{R}$ de l'équation $ax^2 + bx + c = 0$.

Exercice 9. *Distance euclidienne*

Ecrire une fonction distance : float * float -> float * float -> float = <fun> qui implémente la distance euclidienne dans $\mathbb{R}^2$.

Exercice 10. *Ordre lexicographique*

L'ordre lexicographique sur $\mathbb{Z}^2$ est défini par :

$(a ; b) \leq (c ; d)$ si $((a < c)$ ou si $((a = c)$ et $(b \leq d))$.

Ecrire une fonction inf_lexico qui implémente ce test.

Exercice 11.

Ecrire des fonctions qui, étant donnés trois entiers x, y et z calculent ...

→ la somme x + y + z ;
→ la somme des produits deux à deux xy + yz + zx ;
→ le produit xyz ;
→ le triplet (x + y + z; xy + yz + zx; xyz) ;
→ le plus grand des nombres x et y ;
→ le plus grand des nombres x, y et z ;
→ le plus grand des nombres x + y + z, xy + yz + zx et xyz.

Exercice 12.

Prévoir le résultat de l'évaluation de :

```
let f x = x + 1 and g x = 2*x in
let u a b x = a (b x) in
let h = u f g in
h 4;;
```


3. Fonctions anonymes.

Signalons enfin qu'il n'est pas nécessaire de **nommer** une fonction pour s'en servir (☞ fonction anonyme) :

```
(function x -> 2*x) 5;;
- : int = 10
(fun x y -> x*y) 2 5;;
- : int = 10
```

En résumé :

Déclaration d'une fonction anonyme à un argument `(function argument -> expression)`

Déclaration d'une fonction anonyme à plusieurs arguments `fun argument _1 argument_2… -> expression)`

Remarque : Un argument d'une fonction peut être lui-même une fonction !

```
let compo f g x = f(g(x));;
compo (function x -> x+1) (function x -> x*x) 3;;
```

Exercice 13.

Déterminer la réponse de caml aux requêtes suivantes :

→ `let iter f = fun x -> f(f x) ;;`
→ `let succ n = n+1;;`
→ `iter succ;;`
→ `iter succ 3;;`

4. Fonctions curryfiées/décurryfiées.

Reprenons la fonction *produit* précédente (à deux arguments : deux entiers) :

```
let produit x y = x*y;;
produit : int -> int -> int = <fun>
```

Nous aurions pu aussi l'écrire sous la forme suivante (à un seul argument : un couple d'entiers) :

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

On dit que *produit* est la version **curryfiée** de *Produit*.

Le procédé par lequel on passe de *Produit* à *produit* s'appelle la **curryfication**.

L'intérêt des fonctions curryfiées, outre le fait qu'elles sont applicables partiellement, est qu'elle sont en général plus rapides.

En résumé :

Version currifyée d'une fonction `let nom_fonction argument_1 argument_2 … = expression`

Version décurrifyée d'une fonction `let nom_fonction (argument_1, argument_2, … ) = expression`

5. Le type unit.

Une fonction peut très bien n'agir que par **effet de bord**, et ne retourner aucun résultat (c'est l'analogue d'une procédure du Pascal). Par exemple, *print_int* affiche l'entier qui lui est passé en argument :

```
print_int (1+2+3+4);;
10  - : unit = ()
```

Pour des raisons de cohérences, une telle fonction renvoie tout de même un résultat.
Le type de ce résultat est **unit** ; il ne comporte qu'une seule valeur, notée **()**.

Certaines fonctions, quant à elles, n'acceptent aucun argument, par exemple la fonction *sys__time* qui mesure le temps écoulé depuis le début de la session en cours.

6. Le polymorphisme

Les opérateurs de comparaison par exemple sont polymorphes. Qu'entend-on vraiment par cela ? Vous vous rappelez que tout objet Caml a un type, et que toute fonction est un objet : toute fonction a donc un type. Cependant, le type de certaines fonctions (ou d'autres objets) n'est pas toujours figé.

Si on veut écrire la version préfixe de < par exemple, Caml répond :

```
let inferieur_strict x y = x < y ;;
inferieur_strict : ' a –> ' a –> bool = <fun>
```

Il signale ainsi que l'important est que x et y aient le même type, qu'il note 'a (et qui est un paramètre de type), mais sans préciser ce type.
C'est pourquoi Caml répond sans broncher aux requêtes suivantes :

```
4 < 2;;
5.3 < 2.1;;
`` petit " < ``grand ";;
```

Exercice 14.
Donner le type des fonctions :
→ let evalue_en_2 f = f 2 ;
→ let evalue x f = f x .

VII. Le Filtrage

C'est l'un des traits les plus agréables de Caml. Le filtrage permet souvent des définitions très claires et simples de fonctions. Supposons que nous voulions définir une fonction f. Les syntaxes suivantes sont possibles

1. Notion de filtrage

Le **filtrage par motifs** permet d'effectuer un traitement différencié : on confronte la valeur d'une expression *m* à une succession de **motifs** m_1, …, m_n, jusqu'à ce qu'une correspondance soit trouvée : c'est le **filtrage par motifs** ; il sert à reconnaître la forme de cette valeur et permet d'orienter le calcul en associant à chaque motif une expression à évaluer.

Il évite l'emploi de *if then else* imbriqués.

On y recourt beaucoup pour implémenter des fonctions définies par cas et sous-cas.

Il est défini par le mot-clé ***match*** ; La construction est la suivante :

```
match m with
  | m1 -> e1
    …
  | mn -> en
```

Règles.

→ Tous les motifs m_i doivent être du même type t_1.
→ Toutes les expressions e_i doivent être de même type 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.
→ Le motif "joker" _ (symbole de soulignement) placé en dernier a le sens de "dans tous les autres cas".

Exemple :

```
let n = 5;;
n : int = 5
match n with
  |0 -> "nul"
  |_ -> "non nul";;
- : string = "non nul"
```

Remarque : les motifs m_i sont des valeurs ou des identificateurs mais pas déjà définis (tout identificateur dans un filtre est un nouvel identificateur).

Remarque : Caml prévient si le filtrage n'est pas exhaustif, et il prévient également si un cas de filtrage est inutile.

En résumé :

Filtrage d'une expression `match expression with |motif_1 -> expression_1 |motif_2 -> expression_2 …`

2. Filtrage avec gardes

Le filtrage avec gardes correspond à l'évaluation d'une expression conditionnelle - introduite à l'aide du mot clé **when** - immédiatement après le filtrage d'un motif. Si cette expression renvoie *true*, alors l'expression associée au motif est évaluée, sinon le filtrage continue avec le motif suivant.

Exemple :

```
match (x , y) with
 | (0 , _) -> “abscisse nulle”
 | (_ , 0) -> “ordonnée nulle”
 | ( x , y) when x = y -> “abscisse = ordonnée”   (* le motif (x , x) n’est pas admis ! *)
 | _ -> “quelconque”;;
```

En résumé :

Filtrage avec gardes `motif when condition -> expression`

3. Fonctions définies par filtrage

Le filtrage fournit un moyen élégant de définir une fonction : la définition par cas.

Si m_1, …, m_n sont des motifs et e_1, …, e_n sont des expressions, l'expression : *fun(ction)* m_1 -> e_1 | … | m_n -> e_n définit une fonction qui appliquée à un argument *x (y …)* rend la valeur de l'expression e_i associée au premier filtre m_i qui filtre *x (y …)*, en essayant successivement les filtres m_1, …, m_n dans cet ordre.

Pour des raisons de symétrie, on peut placer un "|" devant le premier cas.

Par exemple :

```
let nul_ou_non_nul = function
 | 0 -> "nul"
```

```
  | _ -> "non nul";;
let nul_ou_non_nul x =
 match x with
```

```
| 0 -> "nul"
| _ -> "non nul";;
```

Autres exemples :

- la fonction booléenne OU (en filtrant selon une seule variable) :

```
let ou_logique a b =
match a with
  | true –> true
  | false –> b ;;
ou_logique : bool –> bool –> bool = <fun>
ou_logique false true ;;
– : bool = true
```

- la fonction booléenne ET :

```
let et p q =
match (p,q) with
  | (true,true) -> true
  | _ -> false;;
et : bool -> bool -> bool = <fun>
```

ou encore plus concis :

→ version curryfiée :

```
let et = fun
  | true true -> true
  | _ _ -> false;;
et : bool -> bool -> bool = <fun>
```

→ version non curryfiée :

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

En résumé :

Filtrage argument(s) d'une fonction `fun(ction) |motif_1 -> expression_1 |motif_2 -> expression_2 …`

Exercice 15.

1. Examiner la définition suivante, et indiquer si le filtrage est exhaustif :
   let h = function (5,_) -> 0 | (_,5) -> 1 | (x,9) -> x | (x,y) -> y ;;
2. Dans la suite de la même session indiquer le type et la valeur des expressions :
   → h(5,5), h(0,5), h(5,9), h(9,5) ;; → h(9,4), h(4,9), h(11,9), h(7,8) ;;

Exercice 16.

Ecrire une fonction chiffre : int -> string = <fun> qui affiche en toute lettre le chiffre passé en argument.

VIII. Les exceptions

1. De quoi s'agit-il ?

Une **exception** est par définition un événement qui se produit rarement.
Les exceptions permettent de traiter les situations qui empêchent l'accomplissement normal d'une action.

```
1/0;;
Uncaught exception: Division_by_zero
hd [];;
Uncaught exception: Failure "hd"
v.(5) where v = [|0;1;2;3;4|];;
Uncaught exception: Invalid_argument "vect_item"
```

Caml intègre un mécanisme permettant de produire (on dit aussi "lever", ou "*raise*" en anglais) et d'intercepter (on dit aussi "rattraper", ou "*catch*" en anglais) des "**exceptions**".

→ Lever une exception, c'est signaler qu'une situation anormale est survenue,

→ Rattraper/Traiter une exception, c'est répondre à cette situation en exécutant les actions appropriées.

L'idée est la suivante : lorsqu'une exception se produit, le calcul en cours est interrompu et l'exception est propagée jusqu'à ce qu'elle soit éventuellement rattrapée. Si elle n'est pas rattrapée ("*uncaught exception*"), un message d'erreur est affiché (comme dans les trois exemples ci-dessus).

Les exceptions appartiennent à un type spécial (le type *exn*), qui comporte des constantes prédéfinies parmi lesquelles:

```
Division_by_zero;;
- : exn = Division_by_zero
Out_of_memory;;
- : exn = Out_of_memory
```

Le type *exn* est "extensible", au sens où le programmeur peut définir ses propres exceptions :

```
exception nombre_trop_grand;;
exception nombre_trop_grand defined.
```

La fonction suivante renvoie une exception ; son paramètre de type string permet d'indiquer la nature du problème rencontré :

```
Failure;;
- : string -> exn = <fun>
```

2. Comment lever une exception ?

La fonction *raise* effectue cette tâche :

```
raise;;
- : exn -> 'a = <fun>
```

La fonction *failwith* permet une syntaxe simplifiée : l'expression *failwith* "catastrophe" équivaut à *raise*(*Failure* "catastrophe").

On aura noté que la fonction *raise* renvoie un type polymorphe : cela lui permet de s'adapter à tous les contextes, comme le montrent les deux exemples suivants :

```
let f = function
  | 0. -> failwith "0 : valeur interdite !"
  | x -> 1. /. x;;
f : float -> float = <fun>
f(0.);;
Uncaught exception: Failure "0 : valeur interdite !"
```

```
let rpo c s = (* Rang de la Première Occurrence du caractère c dans la chaine s *)
  let i = ref 0 and found = ref (false) and n = string_length s in
  while (not !found) & (!i < n) do
    if s.[!i] = c then found := true else incr i
  done;
  if !i = n then failwith "Not found" else !i ;;
rpo : char -> string -> int = <fun>
```

Dans le premier de ces deux exemples, il FAUT que l'expression *failwith* "0 : valeur interdite !" soit de type *float*.
Dans le second, il FAUT que l'expression *failwith* "Not found" soit de type *int*.

Il en existe d'autres : Invalid_argument, Match_failure, ...

En résumé :

Définir une exception `exception exception ;;` ou `Failure "exception" ;;`

Lever une exception `raise exception ;;` ou `failwith "exception" ;;`

Exercice 17.

Ecrire une fonction `pred : int -> int = <fun>` qui renvoie l'entier qui précède le cas échéant, et sinon lève l'exception "sans_prédécesseur".

Exercice 18.

Définir une fonction taux de type `: (float -> floa)t -> float -> float = <fun>` associant le taux d'accroissement de f entre x et y.

3. .Comment rattraper une exception ?

Avec la construction ***try*** ... ***with*** ... dont la syntaxe est la suivante :

```
try e with
  | m1 -> e1
  ...
  | mn -> en
```

Les m_i sont des motifs de type exn et les e_i sont des expressions ayant toutes le même type, à savoir celui de e.

Si l'évaluation de e ne déclenche pas d'exception, la valeur de l'ensemble de l'expression est celle de e.
Si en revanche une exception se produit, alors les motifs sont essayés l'un après l'autre : dès que l'un d'eux – disons m_i – filtre cette exception, c'est la valeur de e_i qui est retenue.

Prenons l'exemple de la fonction f définie à la section précédente $f : x \longmapsto 1/x$;

f n'est pas définie en 0, mais la fonction $g : x \longmapsto \sin(x) \times f(x)$ est prolongeable par continuité en 0 et la valeur de prolongement est 1. Voici comment on peut définir g en Caml :

```
let g x =
try sin(x) *. f(x) with
  | Failure "0 : valeur interdite !"-> 1.;;
g : float -> float = <fun>
```



```
#g(0.);;
- : float = 1.0
```

Considérons maintenant l'exemple de la fonction *rpo* définie précédemment et définissons une fonction *subs* telle que *subs src s r* effectue le remplacement de la première occurrence dans *s* de chaque caractère de *src* par le caractère *r* :

```
let subs src s r =
  let n = string_length src in
   for i = 0 to n-1 do  s.[rpo src.[i] s] <- r  done;
  s;;
subs : string -> string -> char -> string = <fun>
subs "lune" "Lorsqu'on lui montre la lune, l'imbécile regarde le doigt" `X`;;
- : string = "LorsqX'oX Xui montrX la lune, l'imbécile regarde le doigt"
subs "xyz" "Lorsqu'on lui montre la lune, l'imbécile regarde le doigt" `X`;;
Uncaught exception: Failure "Not found"
```

Pour rattraper le coup, on modifie comme suit la fonction *subs* :

```
let subs src s r =
  let n = string_length src in
   for i = 0 to n-1 do
    try s.[rpo src.[i] s] <- r with
        | Failure "Not found" -> ()
   done;
s;;
subs : string -> string -> char -> string = <fun>
subs "xyz" "Lorsqu'on lui montre la lune, l'imbécile regarde le doigt" `X`;;
- : string = "Lorsqu'on lui montre la lune, l'imbécile regarde le doigt"
```

En résumé :

Rattraper une exception `try expression with |exception_1 -> expression_1 | ...`