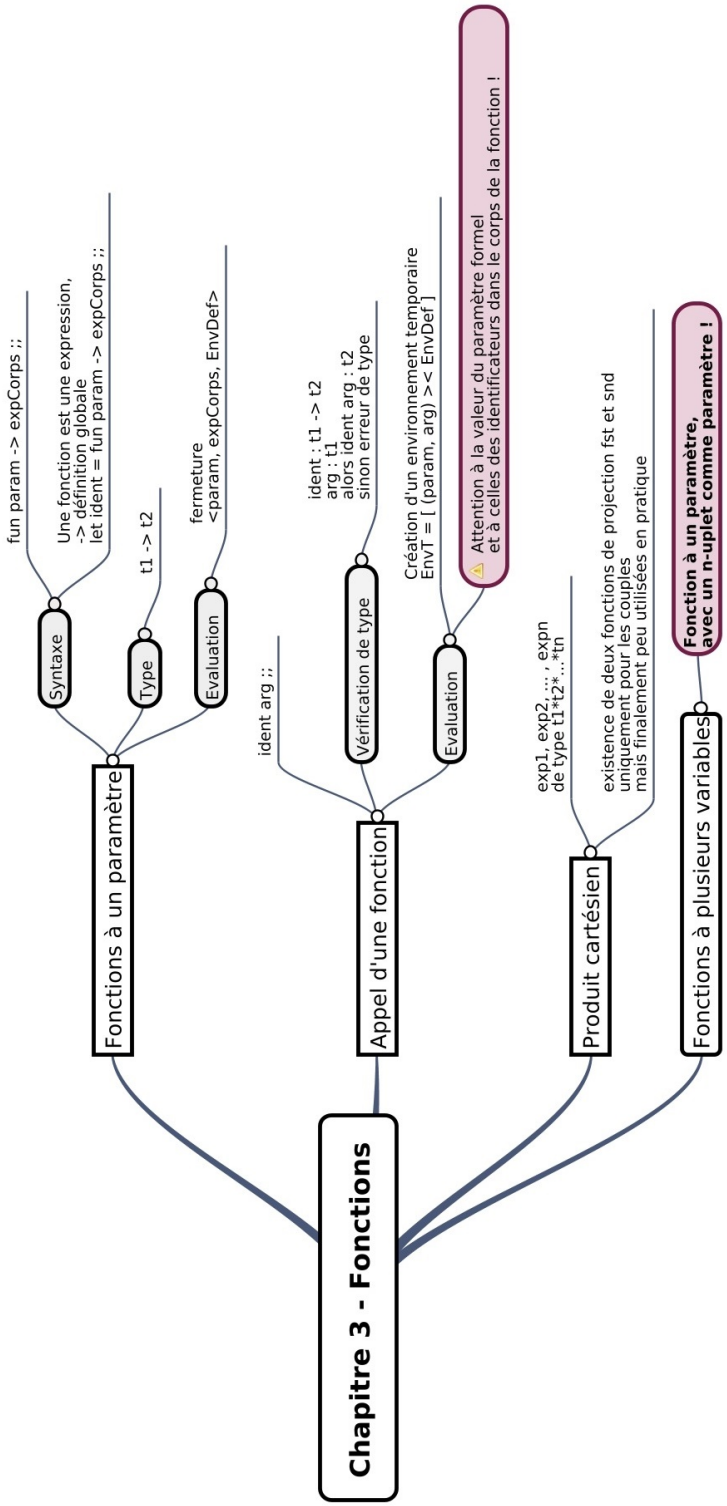
Institut National Universitaire Champollion

CHAPITRE 3 : FONCTIONS

En bref



1. Fonctions à un paramètre

En Caml, la valeur de la fonction en un point x appelé **paramètre** de la fonction est définie par une abstraction, c'est à dire, une expression utilisant x comme **paramètre formel**.
Nous distinguons deux phases dans l'utilisation des fonctions :

- leur **définition**
- leur utilisation, appelé **l'appel de la fonction**

1.1. Syntaxe et exemples

Définition 1

Une fonction est une expression définie par la syntaxe suivante :

```
function param -> ExpCorps
```

Exemple 1

```
function x -> x * x  ; ;
```

Une fonction est **une expression** et nous en verrons des utilisations en tant que telle, mais, dans une écrasante majorité des cas, nous les utiliserons au sein de définitions globales de fonctions, sous la forme :

```
let identf = function param -> ExpCorps
```

Exemple 2

```
let carre = function x -> x*x  ; ;
```

Une définition de fonction utilise les objets suivants :

- l'identificateur `identf` est le **nom** de la fonction
- `param` (x dan l'exemple), est le paramètre formel permettant de définir la fonction
- l'expression `ExpCorps` est une abstraction dépendant du paramètre `param`, permettant de calculer la valeur de la fonction au point `param`. Nous l'appelons le **corps de la fonction**.

1.2. Appel de fonction

Définition 2

Nous appelons une fonction en utilisant la syntaxe suivante :

```
identf arg
```

Où `identf` est le nom de la fonction, et `arg` une expression appelée *paramètre effectif* ou **argument** de la fonction.

Exemple 3

```
carre 2  ; ;
-  : int = 4
```

Remarque : Les parenthèses ne sont pas nécessaires, toutefois, nous vous conseillons de les mettre pour éviter toute ambiguïté !

Exemple 4

```
# carre 1 + 1  ; ;
- : int = 2
# carre (1+1)  ; ;
- : int = 4

(* L'appel de fonction est prioritaire sur la plupart des opérateurs comme + *)

>carre carre 2  ; ;
(* Error  : This function has type int -> int
It is applied to too many arguments*)
(* Ici OCaml pense que carre (le 2ême) et 2 sont les arguments de carre (le 1er)
 et nous explique donc, que nous donnons trop d'arguments puisque la fonction carre
 n'en prend qu'un*)

# carre (carre 2)  ; ;
- : int = 16

(* L'appel de fonction est prioritaire à gauche *)
```

1.3. Type d'une fonction

Comme tout objet en Caml, les fonctions ont un type. De manière générale, le type d'une fonction est de la forme :

$$t_1 \rightarrow t_2$$

Ce qui signifie qu'à un paramètre de type t_1, la fonction associe un résultat de type t_2.
Nous verrons plus en détails au S6, l'algorithme qui permet à Caml de calculer ce type. Mais, dans les grandes lignes, le système analyse le corps de la fonction et en déduit des contraintes sur le paramètre et le résultat. Il y a alors trois cas possibles :

- Dans le cas où l'analyse n'impose pas de contraintes contradictoires mais suffisamment de contraintes pour déterminer parfaitement le type du paramètre et celui du résultat.

  Exemple 5

  ```
  let estNul = function x -> x = 0  ; ;
  ```

  = est un opérateur qui compare deux objets de même type et fournit un résultat booléen, 0 étant un entier, $x = 0$ n'a de sens que si le paramètre x est entier et le résultat est alors un booléen. La fonction est donc de type : `int -> bool`

- S'il y a des contraintes contradictoires, le système arrête alors l'évaluation et affiche une erreur de type.

Exemple 6

```
let contrad = function x -> (x+1) & x  ; ;
```

$x+1$ n'a de sens que si x est un entier, mais (..)&x n'a de sens que si x est un booléen. Il y a donc contradiction.

– Si les contraintes ne sont pas contradictoires, mais sont insuffisantes pour déterminer le type de la fonction. Une telle fonction est appelée **fonction polymorphe** et Caml utilise les symboles `'a, 'b, 'c` etc. pour noter des types quelconques.

Exemple 7

```
let identite = fun x -> x  ; ;
```

Le corps de cette fonction n'impose aucune contrainte sur le type de x, mais seulement que le résultat soit du même type que l'argument. Le type est alors noté : `'a -> 'a`.

Exercice . TD - Exercice A

Donner, lorsque c'est possible, le type des fonctions suivantes :

1. `let f1 = function x -> x +. 1.  ; ;`
2. `lef f2 = function x -> x = true  ; ;`
3. `let f3 = function x -> x ^ " Hello World"  ; ;`
4. `let f4 = function x -> 3  ; ;`
5. `let f5 = function x -> x && (x + 1 = 3)  ; ;`

Exercice . TP - Exercice 1

Écrire les fonctions suivantes :

1. `pair` de type `int -> bool` qui a un entier n associe un booléen valant vrai ssi n est pair.
2. `minus` de type `char -> bool` qui, vaut vrai si le paramètre c de type caractère est une lettre minuscule.

Exercice . TP - Exercice 2

Écrire les fonctions suivantes :

1. `troncature` de type *float -> int*, qui à un réel x, associe l'entier obtenu en débarassant x de ses décimales. *(Indice : Chapitre précédent...)*

```
# troncature 42.45  ; ;
-  : int = 42
# troncature (-42.45)  ; ;
-  : int = -42
```

2. `decimales` de type *float -> float*, qui à un réel x associe sa partie décimale (Attention

aux nombres négatifs !).
(On pourra utiliser la fonction valeur absolue abs_float.)

```
# decimales(-42.45)  ; ;
-  : float = 0.45
```

3. `partie_entiere` de type *float -> int*, qui au réel x, associe le plus grand entier inférieur à x. (La partie entière de -3.2 est -4, celle de -3.00 est -3 et celle de 5.3 est 5.)
La partie entière d'un réel x est égal à sa troncature si $x - troncature(x)$ est positif ou nul et à sa troncature moins 1 sinon.

```
# partie_entiere(-5.61) ; ;
-  : int = -6
```

4. `plus_proche_entier` de type *float -> int*, qui au réel x associe l'entier le plus proche.
On pourra noter qu'il s'agit de la partie entière de $x + 0.5$.

```
# plus_proche_entier(-5.52) ; ;
-  : int = -6
```

5. `arrondi` de type *float -> float*, qui à x associe le réel à deux décimales le plus proche du réel x.
Il faut utiliser la fonction précédente, bien entendu.

```
# arrondi(52.6543) ; ;
-  : float = 52.65
```

6. `francs_en_euros` de type *float -> float*, permettant de convertir en euros une somme exprimée en francs français. On rappelle que le taux de conversion est de 1 euros pour 6.55957 francs.
Améliorer la présentation en arrondissant le résultat au cent inférieur (ce qui revient à arrondir à deux décimales).

Exercice . TP - Exercice 3

Nous voulons mettre une heure donnée sous forme hh.mm sous une forme plus conviviale.

1. Écrire deux fonctions `heures : float -> int` et `minutes : float -> int` telles que, par exemple,

```
#heures 23.42  ; ;
-  : int = 23
#minutes 23.42  ; ;
-  : int = 42
```

2. Écrire une fonction `quelle_heure_est_il : float -> string` qui réalise ce type de transformation. Par exemple :

```
#quelle_heure_est_il(14.45) ; ;
```

```
-  : string = "Il est 14 heure 45"
#quelle_heure_est_il(12.0) ; ;
-  : string = "Il est  midi  pile."
#quelle_heure_est_il(0.34) ; ;
-  : string = "Il est  minuit 34"
```

1.4. Évaluation d'une fonction

Lors de la définition d'une fonction d'une fonction, la réplique Caml contient le mot `<fun>` dans le champ des valeurs. Mais quelle est en fait la valeur associée par Caml à une fonction ?

Définition 3

La valeur associée à une fonction est un triplet appelé **fermeture** composé du paramètre, du corps de la fonction ET de l'**environnement de définition**, c'est à dire l'environnement actif au moment de la définition de la fonction.
La fermeture d'une fonction (sa valeur Caml) sera notée :

```
< param, exp_corps, Env_def >
```

Exemple 8

Si, partant de l'environnement `Env0`, nous écrivons :

```
# let c = 2  ; ;
(* Env1 = [(c,2) >< Env0 ] *)
# let f = function x -> x * c  ; ;
```

La fermeture de f sera

```
< x , x*x , [(c,2) >< Env0 ] >
```

En fait, Caml ne réduit pas une fonction à son corps, mais il "mémorise" aussi l'environnement dans lequel elle a été définie. Voici pourquoi dans le paragraphe suivant.

1.5. Appel de fonction

Lors de l'appel d'une fonction, le système commence (comme toujours) par faire une vérification de type, puis un calcul de valeur.

- **Vérification de type -** Si la fonction est de type $t_1 \to t_2$ et que l'argument est de type t_1 alors le résultat de la fonction sera de type t_2 sinon il y aura une erreur de type.
- **Calcul de la valeur -** Un environnement temporaire est créé : il contient une liaison entre le paramètre formel de la fonction et son argument, cette liaison s'ajoute à l'environnement actif *au moment de la définition de la fonction* (c'est à dire, celui qui a été mémorisé dans la fermeture de la fonction).

  ```
  EnvT = [ (param, arg) >< Env_def ]
  ```

  L'expression du corps de la fonction est évaluée dans cet environnement, c'est la valeur de l'appel de la fonction, puis celui-ci est détruit.

Attention ! Le paramètre formel est remplacé par la valeur de l'argument *au moment de*

l'appel alors que les identificateurs sont remplacés par la valeur à laquelle ils étaient liés *au moment de la définition de la fonction*. C'est précisément pour se rappeler de ces liaisons que la fermeture de la fonction "mémorise" l'environnement qui existait au moment de la définition de la fonction.

Exemple 9

```
(* Env0 = [] *)

# let c = 2  ; ;
(* c : int = 2       Env1 = [(c,2) >< Env0 ]  *)

# let f = function x -> x*c  ; ;
(* f  : int -> int = <fun>         Env2 = [ (f, <x, x*c, Env1 >) >< Env1]   *)

# let c = 5  ; ;
(* c  : int = 5      Env3 = [ (c,5) >< Env2]  *)

#f 3  ; ;
(* EnvT = [ (x,3) >< Env1 ] = [ (x,3), (c,2)]   *)
(* -  : int = 6 *)
```

Exercice . TD - Exercice B

Pour A, B et C, on suppose que l'environnement initial $Env0$ est vide. Pour chaque requête formulée au cours de la session suivante, donnez la réplique du système Caml, et décrire précisément l'évolution de l'environnement.

A.

```
let x = 3  ; ;
let y = 4  ; ;
let f = function x -> 3*x + y*2  ; ;
let x = 2 in y = x+1  ; ;
f 4  ; ;
f x  ; ;
```

B.

```
let f = function a -> let b = 2 in a * b  ; ;
let b = 3 and x = 2  ; ;
f(x)  ; ;
f(b)  ; ;
```

C.

```
let a = 2  ; ;
let b = 12 in 2*b + 4  ; ;
let f = function x -> 2*x + a  ; ;
f(a)  ; ;
```

2. Produit cartésien et fonctions à plusieurs variables

2.1. Produit cartésien

Quelques rappels de Mathématiques...

- Soit A, B des ensembles, le **produit cartésien** $A \times B$ est l'ensemble

$$A \times B = \{(a,b), a \in A, b \in B\}$$

  Un élément (a,b) est appelé **couple**.
- Plus généralement, si $A_1, A_2, \ldots A_n$ sont n ensembles,

$$A_1 \times A_2 \times \ldots \times A_n = \{(a_1, a_2, \ldots, a_n), \forall i \in [\![1,n]\!], a_i \in A_i\}$$

  $(a_1, \ldots, a_n)$ est appelé **n - uplet** et $\forall i \in [\![1,n]\!], a_i$ est la i-ème composante du n-uplet.
  $\forall i \in [\![1,n]\!]$, on définit la i-ème **projection** :

$$p_i : \begin{cases} A_1 \times \ldots \times A_n \to A_1 \\ (a_1, \ldots, a_n) \mapsto a_i \end{cases}$$

En Caml... Le produit cartésien des types t_1 et t_2 est noté $t_1 * t_2$. Plus généralement, le produit cartésien de n types $t_1, \ldots, t_n$ est noté

$$t_1 * t_2 * \ldots * t_n$$

Comme en mathématiques, le constructeur de n-uplets est la virgule. Un n-uplet est donc une expression de la forme

```
exp1, exp2, ... , expn
```

Remarque : Les parenthèses ne sont donc pas nécessaires en Caml, mais il est conseillé e les conserver pour améliorer la lisibilité, pour conserver l'habitude prise en mathématiques et surtout pour se protéger des mauvaises surprises dues à la faible priorité de la virgule face aux autres opérateurs.

Exemple 10

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

# 1, true ;;
(*- : int*bool = 1, true *)

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

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

Attention aux deux derniers ! Observez bien la différence de types obtenus. 1,2,3 est un triplet d'entiers, alors que 1,(2,3) est un couple dont la première composante est un entier et la seconde un couple d'entiers.

Typage et évaluation d'un produit cartésien Sans grande surprise,

- **Typage :** Soit à typer l'expression `exp` définie par $exp_1, exp_2, ..., exp_n$. Si pour tout $i \in [\![1,n]\!]$, exp_i est une expression de type t_i, alors `exp` est de type produit cartésien $t_1 * ... * t_n$
- **Evaluation :** Si pour tout $i \in [\![1,n]\!]$, la valeur de l'expression exp_i dans l'environnement `Env` est v_i, alors la valeur de `exp` dans `Env` est $v_1, v_2, \ldots, v_n$.

Exercice . TD - Exercice C

Donner le type et la valeur expressions suivantes :

```
(1,false) ; ;                          ((1,2),3) ; ;
(2.,1+4,1=1,"salut") ; ;               (1,(2,3))=(1,2,3) ; ;
((‘a‘,1),(‘b‘,2),(‘c‘,3)) ; ;          (1,2,3)=(2,1,3) ; ;
(1,2,3) ; ;
(1,(2,3)) ; ;
```

Projections Pour les produits de DEUX ensembles (uniquement), Caml prédéfinit les projections `fst` et `snd` qui donnent respectivement la première et la deuxième composante d'un couple.

Exemple 11

```
# fst (1,2)  ; ;
(* - :int = 1  *)

# snd (1,2)  ; ;
(* - :int = 2 *)

# fst (1,2,3)  ; ;
(* provoque une erreur de type, fst et snd ne marchent que pour les couples  ! *)
```

Exercice . TD - Exercice D

Donner le type et la valeur des expressions suivantes :

```
fst(2,false) ; ;                       fst (1,2), 3  ; ;
snd(2,false) ; ;
fst((1,2),3) ; ;
```

Remarque : Si ces fonctions `fst` et `snd` existent, nous les utiliserons toutefois très peu, comme on le verra dans le chapitre suivant.

2.2. Fonctions à plusieurs variables

Pour le moment, nous n'avions écrit en Caml que des fonctions à un paramètre. Pour être en mesure d'écrire des fonctions de plusieurs variables, il nous suffira de choisir des n-uplets comme paramètre !

Exemple 12

```
let divise = function (n,d) ->
     if d = 0 then false
     else n mod d = 0  ; ;
(*divise  : int * int -> bool = <fun>*)

divise (4,2)  ; ;
(*-  : bool = true*)
```

Exemple 13

Trois façons d'écrire une fonction qui calcule la plus grande composante d'un couple :

- **Méthode 1 :**

  ```
  let max = function c -> if fst(c) > snd(c) then fst(c) else snd(c)  ; ;
  ```

  Cette méthode n'est pas terrible : il y a deux appels à `fst` et `snd`. On calcule deux fois la même chose, pas très efficace !
- **Méthode 2 - Avec une définition locales :**

  ```
  let max = fun c ->
       let x = fst(c) and y= snd(c) in
       if x>y then x else y  ; ;
  ```

- **Méthode 3 :** La meilleure méthode consiste à utiliser le produit cartésien, évitant ainsi tout appel à `fst` et `snd` .

  ```
  let max = fun (x,y) -> if x>y then x else y  ; ;
  ```

Exercice . TP - Exercice 4

Écrire les fonctions suivantes :

- `min : int*int -> int` qui calcule le minimum de deux entiers
- `norme : int*int*int -> int` qui calcule la somme des carrés des éléments d'un triplet d'entiers. Modifier ensuite votre fonction pour définir localement une fonction carrée.

Exercice . TP - Exercice 5

Écrire une fonction `reel` qui à un réel x et à deux entiers a et b associe le réel dont la troncature est le maximum de a et b (calculé obligatoirement dans une définition locale) et qui a même partie décimale que x.
On pourra utiliser la fonction abs_float

```
# reel(4,5,-4.123)  ; ;
-  : float = 5.123
# reel(7,5,4.123)  ; ;
-  : float = 7.123
```

Exercice . TP - Exercice 6

Écrire les fonctions suivantes :

1. `chiffre(n)` : `int -> int` qui à un entier n associe son dernier chiffre
2. `echange(n,p)` : `int*int -> int` qui à deux entiers n et p associe l'entier obtenu en remplaçant le dernier chiffre de n par celui de p.

Exercice . TP - Exercice 7

Écrire les fonctions booléennes qui à une triplet d'entier (a,b,c) associent respectivement vrai si et seulement si :

1. a,b et c ont même valeur
2. a et b sont égaux mais différents de c
3. la valeur de b est strictement comprise entre a et c
4. parmi a,b et c deux valeurs au moins sont identiques
5. parmi a,b et c deux valeurs exactement sont identiques
6. parmi a,b et c deux valeurs au plus sont identiques

Exercice . TP - Exercice 8

On rappelle que pour résoudre l'équation du second degré (E), on calcule le discriminant $\Delta = b^2 - 4ac$ et qu'alors

- Si $\Delta > 0$, (E) admet deux racines réelles distinctes
- Si $\Delta = 0$, (E) admet une racine réelle double
- Si $\Delta < 0$, (E) n'admet pas de racines réelles

Écrire une fonction `nb_sol(a,b,c)` qui calcule le nombre de solutions de l'équation $ax^2 + bx + c = 0$. On utilisera obligatoirement une définition locale pour calculer Δ.