Premiers pas en OCaml

1 Un nouveau langage

OCaml est un langage de programmation développé en France dans les années 1980 principalement pour la recherche et l'industrie, sous le nom "Objective Caml". Il est à la fois adapté à la programmation fonctionnelle, impérative et orientée objet (POO).

> **Remarque :** Cependant, dans le cadre du programme de MP2I, nous utiliserons surtout ce langage de manière fonctionnelle, préférant le C pour la programmation impérative. La POO (objets, structures, méthodes...) est quant à elle hors-programme.

Ce langage peut être compilé et interprété (ce qui permet d'exécuter dynamiquement les programmes grâce à une transformation du code source en code bas niveau), comme suit (en *bash*) :

- syntaxe de compilation : `ocamlc source.ml -o exec`
- syntaxe d'interprétation : `utop`

Lorsque l'on lance `utop`, on peut alors taper l'une des commandes suivantes :

`définition;;`	prend en compte la définition et affiche le type, la valeur et le nom de l'objet ainsi défini
`expression;;`	évalue l'expression et affiche son type et sa valeur
`#use "source.ml";;`	importe les définitions écrites dans le fichier source `source.ml` et évalue les expressions contenues dedans (interprétant donc, ligne à ligne, le fichier source)
`exit 0;;` (ou Ctrl + D)	sortie de `utop`

OCaml est un langage statiquement typé, c'est-à-dire que le type des valeurs et des fonctions est fixé par les définitions dans le code source (en particulier, *avant* l'appel de fonction et l'exécution).

> **Remarque :** Le langage est muni d'un moteur d'inférence de type capable de déterminer le type d'une fonction, d'une valeur ou d'une expression, même quand celui-ci n'est pas précisé, et ceci grâce aux opérateurs, qui sont spécifiques à chaque type (*cf.* plus loin). Néanmoins, on veillera toujours dans ce cours à expliciter ces types.

Enfin, on met des parenthèses autour des sous-expressions.

> **Exemple :** Ainsi, si `f : int -> int -> int` est une fonction préalablement définie, alors "`f 5 6-1`" donne l'évaluation de $f(5,6)-1$ tandis que "`f 5 (6-1)`" donne celle de $f(5,6-1)$.

2 Structure d'un fichier `.ml`

Un code OCaml écrit dans un fichier `.ml` est essentiellement constitué d'une suite de définitions, qui suivent la syntaxe décrite ci-dessous.

Syntaxe – définitions :

• Pour une définition simple, on a :

```
let nom:t = e
```

où `nom` est un identificateur, `t` un type et `e` une expression de type `t`.

• Pour une fonction, on écrira :

```
let nom_f (arg1:t1) ... (argn:tn) : ts =
    expr_f
```

où $\texttt{arg}_1$,...,$\texttt{arg}_n$ et $\texttt{t}_1$,...,$\texttt{t}_n$ sont les arguments et leurs types, $\texttt{t}_s$ est le type de sortie de la fonction, et `expr_f` est une expression de type $\texttt{t}_s$.

Remarque : Dans la définition précédente, le type de `nom_f` est $\texttt{t}_1$ `->` $\texttt{t}_2$ `->` `...` `->` $\texttt{t}_n$ `->` $\texttt{t}_s$.

Ainsi, une valeur peut être vue comme une fonction d'arité nulle, c'est-à-dire qu'elle possède un nom, un type et une valeur. Les fonctions, quant à elles, on toujours une valeur mais qu'on peut qualifier de "fonctionnelle".

3 Expressions en OCaml

3.1 Qu'est-ce qu'une expression ?

Définition (*environnement*) :

À un instant donné, on appelle environnement courant l'ensemble des valeurs, fonctions et types définis au préalable ou prédéfinis dans le langage.

Définition (*expression*) :

Pour un environnement donné, une expression est une suite de caractères à laquelle on peut attribuer un type et une valeur (au sens large$^{(*)}$)

Remarque : $^{(*)}$Par exmple, la fonction $x \mapsto x+1$ est une valeur, fonctionnelle, en tant qu'élément de $\mathbb{N}^{\mathbb{N}}$.

Remarque : Le type et la valeur d'une expression ne la caractérisent pas (prendre par exemple 3 et $1+2$).

3.2 Les constantes

On donne ici quelques exemples des types prédéfinis du langage OCaml, qui constituent donc aussi des types d'expression possibles.

Type	Exemples	Remarques
`int`	`0`, `1`, `-38`	
`float`	`0.`, `0.5`, `12.`, `12.0`	Il faut toujours mettre un point et préciser la partie entière (ne pas écrire "`.5`", par exemple).
`bool`	`true`, `false`	Ce sont les deux seules valeurs de ce type.
`char`	`'a'`, `'3'`, `'\n'`, `'\"'`, `'\\'`, `'\"'`	
`string`	`"coucou"`, `""`	
`unit`	`()`	C'est l'unique valeur de ce type.

Remarque : Les booléens ne fonctionnement pas avec les entiers comme en C.

Remarque : Le type `unit` est l'équivalent du type `void` en C.

3.3 Les variables

Communément appelées (à tort) "variables", les valeurs définies dans l'environnement courant sont des expressions, ceci valant également pour les fonctions.

Exemple : Si on écrit "`let a = 3`", alors dans la suite `a` est une expression.

Remarque : Ces "variables" ressemblent donc plus aux variables mathématiques (muettes,...) que celles informatiques, qui désignent plutôt des espaces réservés en mémoire.

3.4 Les arguments

Dans une expression définissant une fonction, y compris une fonction anyonyme (*cf.* section suivante), il est possible d'utiliser les arguments de cette fonction. Un argument de fonction est donc une expression tant qu'on est dans le corps de la fonction.

Remarque : L'expression définissant la fonction possède donc un type, mais pas encore de valeur au sens d'une valeur dont le type est celui de sortie de la fonction (sa valeur est en fait fonctionnelle, par exemple : `fun x -> x+1` qui est une fonction anonyme).

Remarque : Les expressions qu'on a vues jusqu'ici correspondent à des feuilles d'un arbre de syntaxe.

3.5 Les fonctions anonymes (ou abstractions)

En OCaml, il est possible de décrire une fonction par une simple expression, c'est-à-dire sans la définir, sans faire appel à `let` et sans lui donner de nom. Une telle fonction est dite anonyme.

Syntaxe – fonctions anonymes :

On utilise la syntaxe suivante :

$$\texttt{fun arg}_1\ \texttt{arg}_2\ \texttt{...}\ \texttt{arg}_n\ \texttt{->}\ \texttt{e}$$

où $\texttt{arg}_1$,...,$\texttt{arg}_n$ sont des identificateurs et `e` est une expression qui peut utiliser ces arguments.

Cette expression est de type $\texttt{t}_1$ `->` ... `->` $\texttt{t}_n$ `->` `t` où $\texttt{t}_1$,...,$\texttt{t}_n$ sont les types des arguments et `t` est le type de `e` qu'on infère à partir de `e` (préciser les types n'est donc pas nécessaire).

3.6 Les appels de fonctions

3.6.1 Les opérateurs prédéfinis

Les opérateurs (arithmétiques, booléens, de comparaison) sont des cas particuliers de fonctions prédéfinies dans le langage, c'est pourquoi on les traite dans cette section.

Définition (*notation préfixe, post-fixe, infixe*) :

On parle de notation préfixe (resp. post-fixe) lorsque l'opérateur se place avant (resp. après) ses opérandes. Pour les opérateurs binaires, il existe aussi la notation infixe où les opérandes se placent de part et d'autre des opérateurs.

Exemple : Le moins unaire ($-$) et la négation ($\neg$) sont notés de manière préfixe. La factorielle (!) est notée de façon post-fixe.

Les opérateurs prédéfinis en OCaml sont infixes s'ils sont binaires et préfixes s'ils sont unaires. On les présente dans le tableau ci-dessous :

Type t	Opérateurs binaires pour t	Opérateurs unaires pour t	Type de l'expression composée
`int`	`+`, `-`, `*`, `/`, `mod`	`-`	`int`
`float`	`+.`, `-.`, `*.`, `/.`, `**` (puissance)	`-.`	`float`
`int` ou `float`	`>`, `>=`, `=`, `<`, `<=`, `<>` (“différent de”)		`bool`
`bool`	`&&`, `||`, `=`	`not`	`bool`
`char`			
`string`	`^` (concaténation)		`string`

Remarque : Pour les `float`, il faut utiliser l'opérateur `-.` pour les fonctions (par exemple : `fun x -> -.x`) mais ce n'est pas nécessaire pour les constantes (par exemple : `-12.0`).

Remarque : `**` ne fonctionne pas sur les `int`, mais il existe des opérateurs de conversion.

Remarque : Notons que contrairement au C, les tests d'égalité se font avec un simple `=`. En fait, il existe aussi en OCaml un opérateur `==`, mais celui teste l'égalité physique, c'est-à-dire liée aux adresses dans la mémoire.

On a de façon générale : $\texttt{op}^1\ \texttt{e}_1$, ou bien $\texttt{e}_1\ \texttt{op}^2\ \texttt{e}_2$, où :
- $\texttt{op}^1$ est un opérateur unaire
- $\texttt{op}^2$ est un opérateur binaire
- $\texttt{e}_1$ et $\texttt{e}_2$ sont des expressions de types respectifs $\texttt{t}_1$ et $\texttt{t}_2$.

3.6.2 Les fonctions en général

Syntaxe – appel de fonction :

Pour appeler une fonction `f`, on utilise la syntaxe suivante :

$$\texttt{f}\ \texttt{e}_1\ \texttt{e}_2\ \ldots\ \texttt{e}_n$$

où $\texttt{e}_1$,...,$\texttt{e}_n$ sont des expressions du type attendu pour les arguments de la fonction. L'expression entière est alors du type de sortie de `f`.

Remarque : OCaml évalue l'expression en lisant les arguments de la droite vers la gauche.

3.6.3 L'alternative

Syntaxe – branche conditionnelle et alternative :

Les branches conditionnelles en OCaml suivent la syntaxe suivante :

`if cond then` e_1 `else` e_2

où `cond` est une expression de type `bool` et e_1 et e_2 sont des expressions de même type `t`. Cette expression est de type `t` et de valeur celle de e_1 si la condition vaut `true`, celle de e_2 sinon.

Remarque : Le "`else`" est obligatoire.

3.6.4 Les n-uplets

Syntaxe – tupples :

Un n-uplet ou tupple est de la forme `(`e_1`,`e_2`,...,`e_n`)` où pour tout $i \in [1..n]$, e_i est de type t_i. Cette expression est alors de type t_1 `*` t_2 `* ... *` t_n.

Exemple : "`(3,"a",(1,2))`" est de type `- : int * string * (int * int)`.

3.6.5 Les définitions locales

Syntaxe – définition locale :

Si l'on veut définir localement une variable à l'intérieur d'une expression, on utilise :

`let def in e`

où `let def` suit la syntaxe des définitions et `e` est une expression de type `t` qui peut utiliser les variables définies dans `def`. L'expression globale reste de type `t`.

Cette commande est s'avère utile, notamment lorsque l'on veut déconstruire un tupple, c'est-à-dire récupérer individuellement ses composantes.

Exemple :
```
let couple = (3,4);;
let (c,d) = couple in a+b
```

4 Définition récursive et mutuellement récursive

4.1 Fonctions récursives simples

Dans une déclaration classique de fonction, l'expression `e` peut faire intervenir les arguments de la fonction, a_1 jusqu'à a_n, des fonctions préalablement définies, mais pas la fonction `f` elle-même. Afin de réaliser des appels récursifs sur une fonction, il faut définir celle-ci comme étant une fonction récursive.

Syntaxe – fonction récursive :

Une fonction récursive est définie comme suit :

`let rec f (`a_1`:`t_1`) ... (`a_n`:`t_n`) : `t_s `= e`

où `e` est une expression de type t_s pouvant faire appel à `f`.

Exemple : Voyons deux façons possibles de coder la factorielle de manière récursive.
En récursif non terminal :
```
let rec facto (n:int) : int =
    if n = 0 then 1
    else (facto (n-1)) * n
```

En récursif terminal :
```
let facto (n:int) : int =
    let rec aux nn res =
        if nn = 0 then res
        else aux (nn-1) (res*nn)
    in aux n 1
```

4.2 Fonctions mutuellement récursives

Syntaxe – fonctions mutuellement récursives :

On peut définir des fonctions qui soient récursives entre elles comme suit :

$$\begin{aligned}&\texttt{let rec f}^1\ \texttt{(a}_1^1\texttt{:t}_1^1\texttt{)}\ \ldots\ \texttt{(a}_{n_1}^1\texttt{:t}_{n_1}^1\texttt{)}\ \texttt{:}\ \texttt{t}_s^1\ \texttt{=}\ \texttt{expr}_1\\ &\texttt{and f}^2\ \texttt{(a}_1^2\texttt{:t}_1^2\texttt{)}\ \ldots\ \texttt{(a}_{n_2}^2\texttt{:t}_{n_2}^2\texttt{)}\ \texttt{:}\ \texttt{t}_s^2\ \texttt{=}\ \texttt{expr}_2\\ &\vdots\\ &\texttt{and f}^k\ \texttt{(a}_2^k\texttt{:t}_1^k\texttt{)}\ \ldots\ \texttt{(a}_{n_k}^k\texttt{:t}_{n_k}^k\texttt{)}\ \texttt{:}\ \texttt{t}_s^k\ \texttt{=}\ \texttt{expr}_k\end{aligned}$$

où pour tout $i \in [1..k]$, $\texttt{expr}_i$ est une expression de type $\texttt{t}_s^i$ pouvant faire intervenir $\texttt{f}^j$ pour tout $j \in [1..k]$ et a_r^i pour tout $r \in [1..n_i]$.

5 Définition de types

5.1 Renommer un type

Syntaxe – définition d'un nouveau type :

En OCaml, il est possible de définir des nouveaux types grâce à la syntaxe suivante :

```
type nomtype = expr_de_type
```

où `nomtype` est un identificateur et `expr_de_type` est une expression de type connue.

Exemple :
```
type entier = int;;
type mon_triplet = entier * float * entier;;
```

Les expressions de type peuvent être simples ou composées, on liste dans ce qui suit les principales sortes de type.

5.1.1 Les types simples

On distingue trois sortes de types simples :

- Les types prédéfinis : `int`, `float`, `bool`, `char`, `string`, `unit`.
- Les types préalablement définis : après une définition de type de la forme `type nomtype = expr`, `nomtype` est une expression de type simple.
- Les variables de type : une variable de type est désignée par `'identificateur`.

Exemple : `fun x -> x;;` dans `utop` donne `a' -> 'a = <fun>` : `'a` est une variable de type.

5.1.2 Les types produit cartésien

Les types produit cartésien sont les `t1 * t2 * ... * tn` où `t1`,...,`tn` sont des expressions de type.

Un type produit cartésien désigne donc un type de n-uplet, ressemblant aux structures en C où chaque champ pouvait avoir un type différent, à ceci près qu'ici, les composantes du n-uplet n'ont pas de nom. Il existe néanmoins un autre type appelé type produit (à ne pas confondre avec le type produit cartésien présenté ici), qui permet de nommer les champs (*cf.* plus loin).

Exemple : `type faux_triplet = int * (int * int)`

Remarque : `t1*t2*t3`, `(t1*t2)*t3` et `t1*(t2*t3)` ne représentent pas le même type.

Pour accéder aux différentes composantes d'un n-uplet, on utilise `let in`.

Exemple :

5.1.3 Les types fonctionnels

Le type `te -> ts` désigne le type d'une fonction à un seul argment de type `te`, et dont le type de sortie est `ts`. Plus généralement, si `f` est une fonction de type de sortie `ts` dont les arguments sont de types respectifs `t1`,...,`tn`, alors le type de `f` est : `t1 -> ... -> tn -> ts`.

Exemple :

En OCaml, une fonction de deux arguments est vue comme une fonction qui associé au premier argument une fonction du deuxième argument. Ainsi, `t1->t2->t3` et `t1->(t2->t3)` représentent le même type.

Exemple :

Remarque :

Remarque : Attention en revanche, les types `t1->t2->t3` et `(t1->t2)->t3` sont différents.

5.2 Définir un type paramétré

5.2.1 Type paramétré à un seul paramètre

Syntaxe – type paramétré simple :

On définit un type paramétré comme suit :

```
type 'a nomtype = expr_de_type
```

où `a` et `nomdutype` sont des identificateurs et `expr_de_type` est une expression qui peut faire intervenir la variable `'a`.

Après une telle définition, `t nomdutype` est une expression de type pour n'importe quelle expression de type `t`. À l'oral, on désignera parfois `'a` par "alpha".

Exemple :
```
type 'a quadruplet = 'a*'a*'a*'a;;
let a:bool quadruplet = (true,true,true,false);;
type quad_bool = bool quadruplet;;
```

Remarque : Attention à ne pas oublier de définir le type `'a` après.

5.2.2 Type paramétré à plusieurs paramètres

Syntaxe – type paramétré à plusieurs paramètres :

On les définit comme suit :

```
type ('a1,'a2,...,'an) nomdutype = expr_de_type
```

où a_1,...,a_n et `nomdutype` sont des identificateurs et `expr_de_type` est une expression de type pouvant faire intervenir `'a1`, `'a2`,...,`'an`.

Exemple :

```
type ('a,'b) quadruplet = 'a*'b*'a*'b;;
let a:(float,int) quadruplet = (2.0,-1,3.0,4);;
type quad_bool = (bool,bool) quadruplet;;
type 'd semi_bool_quad = (bool,'d) quadruplet;;
```

5.3 Définir un type somme

5.3.1 Définition du type somme

Syntaxe – type somme :

Un type somme est un type dont on définit préalablement les valeurs possibles. Comme les types simples, il peut être paramétré, ce que l'on obtient avec la syntaxe :

```
type ('a1,'a2,...,'an) typeSomme =
    | Cons1 [of t1]
    | Cons2 [of t2]
      ⋮
    | Consm [of tm]
```

où `Cons`$_1$,...,`Cons`$_m$ sont des identificateurs appelés constructeurs et `t`$_1$,...,`t`$_n$ sont des expressions de type (que l'on met facultativement), qui peuvent faire intervenir `a`$_1$,...,`a`$_n$.

Voici une syntaxe alternative qui met en valeur le caractère facultatif de ces types :

```
type ('a1,'a2,...,'an) typeSomme =
    | Cons_0^1
      ⋮
    | Cons_0^k
    | Cons_1^1 of t^1
      ⋮
    | Cons_1^ℓ of t^ℓ
```

Après une telle définition :

- n'importe quelle expression de la forme `et`$_1$`,et`$_2$`,...,et`$_n$ `typeSomme` est une expression de type si `et`$_1$,...,`et`$_1$ sont des expressions de type
- chaque `Cons`$_0^i$ est une expression dont le type est `('a1,'a2,...,'an) typeSomme`
- chaque `Cons`$_1^j$ `e` est une expression dont le type est `('a1,'a2,...,'an) typeSomme` ou une spécialisation de ce type, si `e` est une expression de type `t`^{j}.

Remarque : Une fonction dont les arguments appartiennent à un type paramétré (donc elle-même de type paramétré) est appelée fonction polymorphe. Lorsque l'un ou plusieurs des paramètres de type sont définis, on parle alors de spécialisation pour cette fonction.

5.3.2 Filtrage associé

Syntaxe – filtrage :

Le filtrage sur une expression issue d'un type somme permet de réaliser des instructions ou retourner des valeurs différentes en fonction de la valeur de cette expression. Ceci se fait en listant de manière exhaustive les cas possibles ainsi que la valeur associée à chaque cas :

```
match e with
    | Cons_0^1 -> e_0^1
      ⋮
    | Cons_0^k -> e_0^k
    | Cons_1^1 val_1 ou var_1-> e_1^1
      ⋮
    | Cons_1^ℓ val_ℓ ou var_ℓ-> e_1^ℓ
```

où `e` est une expression dont le type est `('a1,'a2,...,'an) typeSomme` ou une spécialisation, $\texttt{val}^i$ est une expression de type $\texttt{t}^1$ (une valeur) et $\texttt{var}^i$ est un identificateur (pour une variable).

On peut aussi écrire une syntaxe plus générale :

```
match e with
    | motif -> expr
```

Remarque :
- L'ordre des motifs est important.
- Il ne faut pas qu'il y ait de motif redondant.
- Les motifs doivent couvrir toutes les valeurs possibles du type somme.

Les warnings possibles dans les filtrages sont donc la redondance et la non exhaustivité.

Remarque : On peut n'est pas limité à une seule valeur lorsqu'on traite le cas $\texttt{Cons}_1^i$: on peut aussi le filtrer successivement selon plusieurs valeurs $\texttt{val}^{i,j}$ différentes.

Exemple : On peut modéliser un jeu de cartes à l'aide de types sommes : *cf.* fichier "`carte-bis.ml`".

5.3.3 Un type somme récursif : implémentation de la pile en OCaml

Comme on a le choix, pour chaque constructeur d'un type somme, de préciser ou non un type associé à ce constructeur, il est possible de définir des types sommes récursifs. On illustre cela dans l'exemple suivant, dédié à une implémentation possible de la structure de pile en OCaml.

Voici donc les définitions des fonctions élémentaires :

Définition du type
```
type 'a pile =
    | PileVide
    | PileNonVide of ('a * ('a pile))
```

Création d'un pile vide
```
let cree_pile_vide : 'a pile = PileVide
```

Test de vacuité
```
let est_pile_vide (p:'a pile) : bool =
    match p with
        | PileVide -> true
        | _ -> false
```

Accès au sommet (sans filtrage)

```
let sommet (p:'a pile) : 'a =
    (* hyp : not(est_pile_vide p) *)
    let PileNonVide (elem,pp) = p in elem
```

Accès au sommet (avec filtrage)

```
let sommet_bis (p:'a pile) : 'a =
    (* hyp : not(est_pile_vide p) *)
    match p with
        | PileVide -> failwith "pile vide"
        | PileNonVide (elem,_) -> elem
```

Empilement

```
let empiler (p:'a pile) (elem:'a) : 'a pile =
    PileNonVide (elem,p)
```

Dépilement

```
let depiler (p:'a pile) : 'a pile =
    (* hyp : not(est_pile_vide p) *)
    match p with
        | PileVide -> failwith "pile vide"
        | PileNonVide (_,pp) -> pp
```

À cela, on rajoute les définitions de deux fonctions non élémentaires : l'une permettant de sommer les valeurs d'une pile d'entiers (donc, `int pile` ici), l'autre permettant de concaténer les chaînes contenue dans une pile de chaîne de caractères (`string pile`), la chaîne au sommet de la pile se plaçant au début.

Somme (récursif non terminal)

```
let rec somme (p:int pile) : int =
    match p with
        | PileVide -> 0
        | PileNonVide (e,pp) -> e + (somme pp)
```

Somme (récursif terminal)

```
let somme_rt (p:int pile) : int =
    let rec aux (pp:int pile) (s:int) =
        (* calcule s + la somme des éléments de pp *)
        match pp with
            | PileVide -> s
            | PileNonVide (e,spp) -> aux spp (s+e)
    in aux p 0
```

Concaténation (non temrinal)

```
let rec conca (p:string pile) : string =
    match p with
        | PileVide -> ""
        | PileNonVide (e,pp) -> e^(conca pp)
```

Concaténation (temrinal)

```
let conca_rt (p:string pile) : string =
    let rec aux (pp:string pile) (s:string) =
        match pp with
            | PileVide -> s
            | PileNonVide (e,spp) -> aux spp (s^e)
    in aux p ""
```

Remarque : Remarquons bien que l'opération de concaténation ne se fait pas dans le même sens pour `conca` et `conca_rt`, même si ces deux fonctions sont bien équivalentes (réfléchir à

la façon dont se construit la chaîne totale dans les deux cas).

5.3.4 Le type option : un type somme paramétré prédéfini

Le type option est un type particulier, il est défini comme suit :

```
type 'a option =
    | None
    | Some of 'a
```

Il est notamment utile si l'on veut s'autoriser à ne "rien" retourner dans une fonction : alors, on retournera `Some v` lorsqu'il est possible de trouver une valeur `v` convenable, et `None` dans le cas contraire.

6 Définir un type produit

Les types produit, aussi appelés types enregistrement ou "record", sont l'équivalent des `struct` en C : ils permettent de rassembler des données de types différents et d'y accéder grâce à des noms.

Syntaxe – type produit :

Un type produit peut, tout comme un type somme, être paramétré. On le définit comme suit :

```
type ('a1,'a2,...,'an) typeProduit = {
    champ1 : t1
    champ2 : t2
      ⋮
    champm : tm
}
```

où `champ`$_1$,...,`champ`$_n$ sont des identificateurs et `t`$_1$,...,`t`$_n$ sont des expressions de type pouvant utiliser `a`$_1$,...,`a`$_n$.

• Lorsqu'on définit alors une variable de type `typeProduit`, on adopte la syntaxe suivante :

```
let var : (s1,...,sn) typeProduit = {champ1 = val1,...,champm = valm}
```

où `s`$_1$,...,`s`$_n$ sont des expressions de type et `val`$_1$,...,`val`$_m$ sont de types respectifs `t`$_1$,...,`t`$_m$ (avec la spécialisation induite par `s`$_1$,...,`s`$_n$).

• Pour $i \in [1..m]$, on peut alors accéder au champ `champ`$_i$ de `var` avec : `var.champ`$_i$.

Remarque : Contrairement au C, il est nécessaires de réécrire les noms des champs lors de la définition d'une variable, mais on peut par contre les initialiser dans un ordre différent de celui indiqué dans la définition du type produit.