Le langage fonctionnel CAML

Introduction :

Séance 1 :types de base, définitions, tuples, fonctions, définitions de fonctions, structures if-then-else, priorité des opérateurs.

1.1 Types de base
1.2 Définitions
1.3 Tuples (couples , triplets, n-uplets)
1.4 Fonctions
1.5 Définitions de fonctions
1.6 Structures if-then-else
1.7 Une note sur la priorité des opérateurs

Séance 2 : fonctions récursives simples, récursivité croisée.

2.1 Fonctions récursives simples
2.2 Récursivité croisée

Séance 3 : Fonctions d'ordre supérieur, curryfication, transformation en récursivité terminale

3.1 Fonctions d'ordre supérieur et curryfication
3.2 Transformation en récursivité terminale

Séance 4 : Listes, pattern matching, fonctions polymorphiques

4.1 Listes
4.2 Pattern matching
4.3 Fonctions polymorphiques
4.4 Preuves de correction des fonctions sur les listes
4.5 Transformation en récursivité terminale et pattern matching ????

Séance 5 : Définitions de types

5.1 Définition et utilisation de types énumérés ???
5.2 Définition et utilisation de types sommes ???
5.3 Types sommes récursifs ????
5.4 Définition et utilisation de types génériques ????

INTRODUCTION :

Caml possède un mode interactif où il analyse et répond à chaque phrase entrée par l'utilisateur.

Le caractère # invite l'utilisateur à entrer une phrase écrite dans la syntaxe Caml, phrase qui par exemple nomme une valeur, explicite une définition ou décrit un algorithme. Chaque phrase Caml doit terminer par ;; puis l'utilisateur valide sa frappe par un retour chariot. Dès lors, Caml analyse la phrase, calcule son type (inférence des types), la traduit en langage exécutable (compilation) et enfin l'exécute pour fournir la réponse demandée. En mode interactif, Caml donne systématiquement une réponse qui contient :

- Le nom de la variable déclarée s'il y en a.
- Le type trouvé pendant le typage.
- La valeur calculée après exécution.

Séance 1

Types de base, définitions, tuples, fonctions comme objets de première classe, définitions de fonctions, structures if-then-else, priorité des opérateurs.

1.1/ Types de base

Les types de base gérés par CAML sont au nombre de 5. Il s'agit des entiers (noté *int*), des réels (noté *float*), des booléens (noté *bool*), des chaînes de caractères (noté *string*) et des caractères (noté *char*).

1.1.1/ ENTIERS. Les entiers gérés par CAML sont les entiers mathématiques entre : -1073741824 et 1073741823. Les opérations sur les entiers sont données dans le tableau :

Opérations sur les entiers	
+	addition
-	soustraction et moins unaire
*	multiplication
/	division entière
mod	reste de la division entière

Exemples d'expressions entières :

```
# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
```

1.1.2/ REELS (Flottants). Les opérations sur les réels sont données dans le tableau :

Opérations sur les reéls (flottants)	
+.	addition
-.	soustraction et moins unaire
*.	multiplication
/.	division entière
**	exponentiation

Exemples d'expressions réelles :

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
```

Certaines fonctions prédéfines peuvent être appliqués sur des réels.

Fonctions sur les réels(flottants)	
ceil	Arrondi (retourne un réel)
floor	Partie entière (retourne un réel)
sqrt	Racine carrée
exp	exponontielle
log	Log népérien

Exemple :

```
# ceil 3.4 ;;
- : float = 4
# floor 3.4 ;;
- : float = 3
# exp 1.0;;
- : float = 2.71828182846
# log 2.71828182846;;
- : float = 1.0
```

1.1.3/ BOOLEENS. Les valeurs booléennes s'écrivent true et false en CAML. On peut utiliser la conjonction logique (qui se note && ou &), la disjonction logique (notée or ou ||) et la négation logique (not). Exemples d'expressions booléennes correctes en CAML.

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
# true && false ;;
- : bool = false
# true or false ;;
- : bool = true
```

Il faut noter que la conjonction logique et la disjonction logique sont calculées de manière lazy (synonyme : non-stricte) de gauche à droite en CAML. Cette remarque signifie que, si la valeur de gauche permet à elle seule de calculer la valeur de toute l'expression, ce qui se trouve à droite de la conjonction ou de la disjonction ne sera jamais évalué.

Les opérateurs d'égalité et de comparaisons permettent de construire des expressions booléennes plus élaborées.

Opérateurs d'égalité et de comparaison	
= égalité structurelle	< inférieur
== égalité physique	> supérieur
<> négation de =	<= inférieur ou égal
!= négation de ==	>= supérieur ou égal

Les opérateurs d'égalité et de comparaison sont aussi bien utilisables pour comparer deux entiers que deux chaînes de caractères. La seule contrainte reste que leurs deux opérandes doivent être du même type.

Exemples :

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "un" < "deux" ;;
- : bool = false
# 0 < '0' ;;
Characters 4-7:
This expression has type char but is here used with type int
```

1.1.4/ CHAINES DE CARACTERES. Les chaînes de caractères s'écrivent entre guillemets (à l'intérieur d'une telle chaîne de caractères, le caractère « guillemet » se note \"). La seule opération disponible sur les chaînes de caractères est la concaténation, notée ^ (accent circonflèxe). Exemple de chaîne de caractères en CAML.

```
# "Ceci est une ch" ^ "aîne de caractères !" ;;

- : string = "Ceci est une chaîne de caractères !"
```

1.1.5/ CARACTERES. Les caractères se note entre apostrophes inverses (on peut obtenir une apostrophe inverse en laissant enfoncée la touche ALT du clavier tout en tapant 96 sur le clavier numérique) : Exemple de caractères en CAML.

```
# 'B' ;;
- : char = 'B'
# '&' ;;
- : char = '&'
```

1.2 Définitions

Une définition est une déclaration qui associe un nom à une valeur. On distingue les déclarations globales des déclarations locales. Dans le premier cas, les noms déclarés sont connus de toutes les expressions suivant la déclaration ; dans le second, les noms déclarés

ne sont connus que d'une expression. Il est également possible de déclarer simultanément plusieurs associations nom-valeur.

Déclarations globales :

let *nom* = *expr* ;;

où *nom* représente l'identificateur utilisé et *expr* l'expression qui lui est associée. Exemple:

```
# let x = 7;;
x : int = 7

# x+2;;
- : int = 9
```

Une déclaration de même nom annule la précédente.

Déclarations globales simuultanées:

let $nom_1 = expr_1$
and $nom_2 = expr_2$
:
and $nom_n = expr_n$;;

Une déclaration simultanée déclare au même niveau différents symboles. Ils ne seront connus qu'à la fin de toutes les déclarations. Exemple:

```
# let x = 1 and y = 2 ;;
x : int = 1
y : int = 2

# x + y ;;
- : int = 3
```

Déclarations locales:

let *nom* = $expr_1$ **in** $expr_2$;;

Le nom *nom* n'est connu que pour le calcul de $expr_2$. La déclaration locale lui associe la valeur de $expr_1$. Exemple:

```
# let xl = 3 in xl * xl ;;
- : int = 9
```

1.3 Tuples

A partir des types de base, il est possible de construire des types plus complexes. Dans ce cours, on abordera principalement 3 méthodes de construction permettant d'accéder à des types plus évolués : les tuples (couples, triplets, ...), les fonctions (voir ci-dessous) et les listes (voir Séance 4).

Un tuple se note en écrivant ses composantes les unes à la suite des autres, séparées par des virgules et éventuellement encadrer par des parenthèses afin d'augmenter la lisibilité. Le type d'un tuple est constitué par les types de ses composants séparés par une étoile *.

Exemples :

```
# ( 12 , 20) ;;
- : int * int = 12, 20
# ( 12 , "octobre" ) ;;
- : int * string = 12, "octobre"
# ( 12 , "octobre" , true) ;;
- : int * string * bool = 12, "octobre", true
```

Seuls les couples sont munis de deux fonctions, appelées *fst* et *snd* permettant d'accéder respectivement au premier élément et au second élément du couple. Exemple :

```
# fst ( 12 , "octobre" ) ;;
- : int = 12
# snd ( 12 , "octobre" ) ;;
- : string = "octobre"
```

1.4 Fonctions

1.4.1 Expression fonctionnelle.

Une expression fonctionnelle est constituée d'un *paramètre* et d'un *corps*. Le paramètre formel est un nom de variable et le corps une expression. Elle est introduite par le mot réservé *function* et permet de d'exprimer des fonctions dites anonymes (sans noms).

---

function *p* -> expr

Exemple : la fonction qui élève au carré son argument s'écrit en CAML :

```
# function x -> x*x ;;
- : int -> int = <fun>
```

Le système CAML déduit son type. Il s'agit du type fonctionnel *int -> int* qui indique une fonction attendant un paramètre de type *int* et retournant une valeur de type *int*.

L'application d'une fonction à un argument s'écrit comme la fonction suivie par l'argument.

```
# (function x -> x * x) 5 ;;
- : int = 25
```

Le calcul d'une application revient à calculer le corps de la fonction, ici x * x, où le paramètre formel, x, est remplacé par la valeur de l'argument (ou paramètre d'appel, ou encore

paramètre effectif), ici 5.

Dans la construction d'une expression fonctionnelle, *expr* est une expression quelconque. En particulier, *expr* peut elle-même être une expression fonctionnelle.

```
# function x -> (function y -> 3*x + y) ;;
- : int -> int -> int = <fun>
```

Les parenthèses ne sont pas obligatoires. On peut écrire plus simplement :

```
# function x -> function y -> 3*x + y ;;
- : int -> int -> int = <fun>
```

Le type de cette expression peut être lu de la façon usuelle comme le type d'une fonction qui attend deux entiers et retourne une valeur entière. Mais dans le cadre d'un langage fonctionnel comme Objective CAML il s'agit plus exactement du type d'une fonction qui attend un entier et retourne une valeur fonctionnelle de type *int -> int* :

```
# (function x -> function y -> 3*x + y) 5 ;;
- : int -> int = <fun>
```

On peut, bien entendu, utiliser l'expression fonctionnelle de façon usuelle en l'appliquant à deux arguments. On écrit :

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

Lorsque l'on écrit f a b, il y a un parenthésage implicite à gauche ce qui rend cette expression équivalente à : (f a) b.

Revenons sur le détail de l'application

(function x -> **function** y -> 3*x + y) 4 5

Pour calculer la valeur de cette expression, il faut calculer la valeur de

(function x -> **function** y -> 3*x + y) 4

qui est une *expression fonctionnelle* équivalente à

function y -> 3*4 + y

obtenue en remplaçant x par 4 dans 3*x + y. En appliquant cette valeur (qui est une fonction) à 5 on obtient la valeur finale 3*4+5 = 17 :

```
# (function x -> function y -> 3*x + y) 4 5 ;;
- : int = 17
```

1.4.2 Fonctions à plusières arguments.

Il existe une manière plus compacte d'écrire des expressions fonctionnelles à plusieurs paramètres. C'est une survivance des anciennes versions du langage Caml. Sa forme est la suivante :

fun $p_1 \ldots p_n$ -> expr

Où les pi sont les arguments et expr le corps de la fonction. Elle permet de ne pas répéter le mot clé **function** ni les flèches. Elle est équivalente à la traduction suivante :

function p_1 ->... -> **function** p_n -> expr

```
# fun x y -> 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y -> 3*x + y) 4 5 ;;
- : int = 17
```

1.5 Définitions de fonctions

Comme dans le cas des valeurs simples, il est possible d'assigner un nom à une fonction. Comme le montrent les exemples suivants :

```
# let carre = function x -> x*x ;;
carre : int -> int = <fun>
# carre 5 ;;
- : int = 25
```

```
# let g = function x -> function y -> 2*x + 3*y ;;
g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

```
# let f = fun x y -> 3 * x + y ;;
f : int -> int -> int = <fun>
# f 2 4;;
- : int = 10
```

Pour simplifier l'écriture, la syntaxe suivante est acceptée pour la définition d'une fonction d'arité n (avec n>=1) :

```
# let carre x = x*x ;;
carre : int -> int = <fun>
```

```
# let g x y = 2*x + 3*y ;;
g : int -> int -> int = <fun>
```

```
# let f x y = 3 * x + y ;;
f : int -> int -> int = <fun>
```