[PF5] Cours 1

“

michele.pagani@pps.univ-paris-diderot.fr

Notation :
Session 1: $\frac{1}{2}$ *projet* + $\frac{1}{2}$ *Exam*
Session 2: $max((\frac{1}{2}$ *projet* + $\frac{1}{2}$ *Exam2), Exam2)*

1. Syles de programmation
    1. Impératif
    2. Fonctionnel
2. Compilation
    1. Boucle d'interprétation
    2. Remarque
3. Types
    1. int
        1. Valeur
        2. Opérateurs
        3. Exemple
    2. float
        1. Valeur
        2. Opérateurs
        3. Conversion
        4. Exemple
    3. bool
        1. Valeur
        2. Opérateurs
        3. Exemple
    4. fonctionnels
        1. Exemple
4. Déclaration des valeurs
    1. let
    2. let rec
    3. let in

Syles de programmation

1. Impératif

- **Programme**:
  Séquence structuée d'instruction à la machine
- **Variable**:
  Nom pour un emplacement mémoire qui contien une valeur

- **Affectation**:
  `x = 2 + 1;` Change la valeur de la case memoire de x
- **Usage intessif de l'iteration**

2. Fonctionnel

- **Programme**:
  définition d'une fonction à partir de fonction plus élémentaires
- **Variable**:
  Une inconnue sur un ensemble de valeur
- **Application des fonction**:
  `(fun x -> x+1)2;;` Expression représentant la valeur 3
- **Usage intessif de la recursion**

Compilation

Deux type de fonctionnement:

1. **Compilé** (Java, C) :
   Permet d'obtenir des éxécutables autonomes

```
ocamlc      // compilateur en ligne de code-octet
ocamlrun    // interprète de code-octet
ocamlopt    // compilateur en ligne de code natif
js_of_ocaml //compilateur vers JavaScrip
```

1. **Interpréter** :
   Permet d'observer les requêtes une par une

```
ocaml       // lance une boucle d'interprétation
```

1. Boucle d'interprétation

L'utilisateur tape une **requête**, Une expréssion se termine avec deux `;;`

1. ***Caml*** analyse la **syntaxe**, affiche une erreur si la syntaxe est inccorecte.
2. Si la syntaxe est correcte, l'interpréteur **calcule** le **type**, affiche une erreur si l'expression est mal typée.
3. Si l'expression est bien typée, l'interpréteur **évalue** l'expression, puis **affiche** le **type** et la **valeur** obtenue.

Ceci est une boucle **Read-Eval-Print**

2. Remarque

- Ce sont les `;;` qui termine la requête, pas les saut de ligne. Un saut de ligne dans une requête sont des **espaces** comme les autres
- Les `;;` **ne font pas partie de la syntaxe de *Caml* mais de l'interpréteur**.
- Les commentaires sont entre `(*` et `*)`, et peuvent être sur plusieurs lignes.

Types

- Typés **automatiquement** par l'interpréteur/compilateur
- type de base : `int, float, bool, ...`
  type fonctionnels : `int -> float, (int -> int) -> int` , ...
  ...
  type structurés *(voit plus tard)*
  Objets et classes *(pas dans ce cours)*
- Grande fléxibilité dû à la **polymorphie** *(voir plus tard)*

```
val qSort : 'a list -> 'a list = <fun>
```

- Aussi petits problèmes dues au typage automatique :
  Pas de conversion automatique entre types

1. `int`

1.1. Valeur

`...,-2, -1, 0, 1, 2, ...`

1.2. Opérateurs

- `+, -, *, /, mod, ...` *(infix)*

1.3. Exemple

```
3 + 4;;
- : int = 7

(3 + 4) mod 2;;
- : int = 1
```

2. `float`

2.1. Valeur

`...,-2.0, 3.14, 5e3, 6e-9, ...`

2.2. Opérateurs

- Arithmetiques:
  `+., -., *., /., mod, ...`
  Attetion ! Sur `float` les opérateur arithmetique s'écrivent avec un `.`
- Réels :
  `sin, sqrt, log, floor, ...`

2.3. Conversion

Il y a des fonctions de conversion de typee entre int et float.

```
float_of_int;;
- : int -> float = <fun>

(float);;
- : int -> float = <fun>

int_of_float;;
- : float -> int = <fun>
```

2.4. Exemple

```
sin  (2.0 / .3.0);;
- : float = 0.618369803069737

3.0 +. 2.5;;
- : float = 5.5

3.0 + 2.5;; (*Pas de point apres l\'opérateur*)
  ^^^
Error : This expression has type float but an expression was expected of type int

int_of_float 3.0 + int_of_float 2.5;;
- :  int = 5

3 +. 2.5;; (*Pas de convertion implicite entre int vers float*)
  ^
Error : This expression has type int but an expression was expected of type float

float_of_int  3 +. 2.5;;
- :  float = 5.5
```

3. `bool`

3.1. Valeur

`true, false`

3.2. Opérateurs

- Logiques :
  - **`not`** : négation
  - `&&, &` : et séquentiel (*infix*)
  - `||, or` : ou séquentiel (*infix*)
- Comparaison :
  - `=` : égalité **1 Seul `=` contrairement au `Java ou Python : ==`**
  - `>, >=` : plus grand, plus grand ou égal
  - `<, <=` : plus petit, plus petit ou égal
- conditionnel: `if cond then e1 else e2` :
  - cond est une exprèssion de type `bool`
  - e1 et e2 sont deux expressions de **même type**, qui est aussi le type du conditionnel

- seulement une des deux branche e1, e2 est évaluée.

3.3. Exemple

```
not false && false;;
- : bool = false

not (false && false);;
- : bool = true

true = false;;
- : bool = false

3 = 3 ;;
- : bool = true

4 + 5 > 10;;
- : bool = false

2.0 *. 4.0 > 7.0;;
- : bool = true

if (3 < 4) then 1 else 0;;
- : int = 1

if (4 < 0) then 1 else 0;;
- : int = 0
```

4. `fonctionnels`

Le type d'une fonction n'est plus un type de base.

- Une façon d'introduire les fonctions est à travers:

```
fun var_1 ... var_n -> expr
```

Ou

```
function var_1 ... var_n -> expr
```

- Pour evaluer une fonction il faut lui donner des arguments
- Le resultat de l'evaluation peut être une autre fonction *(évaluation partielle)*

- L'argument d'une fonction peut être une fonction

4.1. Exemple

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

# function x -> x * 2;;
- : int -> int = <fun>

# fun x y  -> x * y;;
- : int -> int -> int = <fun>

# (fun x -> x * 2) 3;;
- : int = 6

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

# (fun x y  -> x * y) 3 2;;
- : int = 6

# fun f  -> (f(f2));;
- : (int -> int) -> int = <fun>

# (fun f  -> f(f x) (fun x -> x * x);;
- : int -> int = <fun>

# (fun f  -> f(f x) (fun x -> x * x) 2 ;;
- : int = 16
```

Déclaration des valeurs

1. let

```
let nom = expr
```

- Associe à `nom` la valeur de l'expression `expr` pour la réutiliser après:

```
# let x = 2 + 3;;
val x : int = 5

# x * 3;;
- : int = 15
```

- En particulier, on peut donner un nom aux fonctions:

```
# let f = (fun x -> x * 2);;
val f : int -> int = <fun>

# f 3;;
- : int = 6
```

- `OCaml` fournit une syntaxe simplifiée:

`let f x y = expr` est équivalent à `let f = fun x y -> expr`

```
# let g f x = f (f ( x * 2));;
val g : (int -> int) -> int -> int = <fun>

# let sq x = x * x;;
val sq : int -> int = <fun>

# let h = g sq;;
val h : int -> int = <fun>

# h 1;;
- : int = 16
```

2. let rec

```
let nom rec = fun arg_1 arg_2 ... -> expr
```

- `let ... rec` permet une définition récursive d'une fonction `nom`

```
# let rec fact = fun x -> if (x = 0) then 1 else x * fact(x-1)
val fact : int -> int = <fun>
```

- Si on utilise simplement `let` on obtient une erreur:

```
# let fact = fun x -> if (x = 0) then 1 else x * fact(x-1)
                                                  ^^^
Error : Unbound value fact
```

`let` définit fact dans tout les expressions qui suivent la déclaration **mais** pas dans l'expression a droite de `->`

- On peut aussi utiliser la syntaxe simplifiée, comme pour `let`

```
# let rec fact = if (x = 0) then 1 else x * fact(x-1)
val fact : int -> int = <fun>
```

3. let in

`let nom = expr_1 in expr_2`

- permet une déclaration locale de nom dans expr_2:

```
# let x = 3 in x + 4;;
- : int = 7
# let y = x + 4;;
         ^^^
Error : Unbound value x
```

- Alors qu'un nom déclaré par `let` ou `let rec` est par contre connu par toute les expressions qui suivent la déclaration :
- Une déclaration locale peut redefinir **localement** un nom glbal

```
# let x = 3;;
val x : int = 3

# let x = 3 in x;;
- : int = 3

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

A FINIR