OCaml

1 - Les bases

http://tsi.tuxfamily.org/OCaml

vincent.maille@ac-amiens.fr

Louis THUILLIER Lycée

7 février 2024

Python (comme le C, C++,...) est un langage **impératif** : on utilise des emplacements/variables, on y met des valeurs à l'aide d'une suite d'instructions, on déplace, modifie ces valeurs ...

Différences avec Python

OCaml est un langage **fonctionnel** : on crée des fonctions, qui sont appelées par d'autres fonctions (comme en mathématiques, on définit des fonctions, qui sont composées entre elles,...)

Exemple avec le calcul de factorielle n

Exemple avec le calcul de factorielle n

$$0! = 1 \text{ et } \forall n \in \mathbb{N},\ n! = \prod_{k=1}^{n} k!$$

En Python :

```
def fact(n) :
    f = 1
    for k in range(2,n+1) :
        f = f * k
    return f
```

Exemple avec le calcul de factorielle n

$$0! = 1 \text{ et } \forall n \in \mathbb{N},\ n! = \prod_{k=1}^{n} k!$$

En Python :

```
def fact(n) :
    f = 1
    for k in range(2,n+1) :
        f = f * k
    return f
```

En OCaml :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1)
;;
```

Exemple avec le calcul de factorielle n

$$0! = 1 \text{ et } \forall n \in \mathbb{N},\ n! = \prod_{k=1}^{n} k!$$

En Python :

```
def fact(n) :
    f = 1
    for k in range(2,n+1) :
        f = f * k
    return f
```

En OCaml :

```
let rec fact = function
  | 0 -> 1
  | n -> n * fact (n-1)
;;
```

$$0! = 1 \text{ et } \forall n \in \mathbb{N}^*,\ n! = n \times (n-1)!$$

Python est un langage **interprété** (il faut un moteur Python qui se charge de lire les lignes une à une au fur et à mesure et de les exécuter).

Différences avec Python

Caml = "Categorical Abstract Machine Language", soit "langage de la machine abstraite catégorique" est un **langage compilé**. Le O de OCaml signifie que c'est un langage **orienté objet**.

Même si OCaml est un langage compilé, on dispose d'un mode interprété, appelé Top-Level

Même si OCaml est un langage compilé, on dispose d'un mode interprété, appelé Top-Level

En réponse au signe d'invite du mode interactif (le caractère #), les **phrases** ou **requêtes** peuvent occuper plusieurs lignes et se terminent par ;;

```
#1+2;;
- : int = 3
```

On peut nommer de manière globale un résultat

```
#let v = 7;;
v : int = 7
#v + 3;;
- : int = 10
```

$$\underbrace{v}_{\text{nom}} : \underbrace{int}_{\text{type}} = \underbrace{7}_{\text{valeur}}$$

Vocabulaire

v est ici un **identificateur global**, c'est à dire accessible dans tout le programme et **immuable** (ou **non mutable**), c'est à dire que l'on ne peut pas le modifier sans le redéfinir. Il correspond au "soit" en mathématiques comme dans « Soit I, le milieu de $[AB]$ ».

Définir un identificateur local :

```
# let x = 14 in x / 3;;
- : int = 4
# x;;
Characters 1-2:
  x;;
  ^
Error: Unbound value x
```

x n'est connu que dans l'expression qui suit le `in` : le message d'erreur : « L'identificateur x n'est pas défini »

Définir un identificateur global à l'aide d'un identificateur local :

```
# let y = let x = 14 in x /3;;
val y : int = 4
# y;;
- : int = 4
# x;;
Characters 1-2:
  x;;
  ^
Error: Unbound value x
```

Définir plusieurs identificateurs simultanément :

```
# let x = 3 and y = 4;;
val x : int = 3
val y : int = 4
```

Ou dans en tant qu'identificateurs locaux :

```
# let x = 3 and y = 4 in x*y;;
- : int = 12
```

Pensez que votre code a vocation à être relu et corrigé aux concours... Mettre des commentaires :

```
let x = 3 (* Ceci est un commentaire en OCaml *);;
```

Définir une fonction :

```
# let f(x)=2*x+1;;
val f : int -> int = <fun>
# f(3);;
- : int = 7
```

Ou en mode "abrégé" :

```
# let f x = 2*x+1;;
val f : int -> int = <fun>
# f 3;;
- : int = 7
```

Définir une fonction à 2 variables :

```
# let perim_Rect (long,larg) = 2 * (long + larg);;
val perim_Rect : int * int -> int = <fun>
# perim_Rect(4,5);;
- : int = 18
```

Ou sans parenthèse (Voir prochain cours...) :

```
# let perim_Rect long larg = 2 * (long + larg);;
val perim_Rect : int -> int -> int = <fun>
# perim_Rect 4 5;;
- : int = 18
```

Ecrire du texte :

```
# print_string "OCaml";;
OCaml- : unit = ()
```

Ecrire un entier :

```
# print_int (5+3);;
8- : unit = ()
```

Aller à la ligne :

```
# print_string "Bonjour,";
print_string "ça va ? ";
print_newline();
print_string "Oui!";;
Bonjour,ça va ?
Oui!- : unit = ()
```

```
let n = -3;;
```

SI .... ALORS ....

```
if n > 2 then
          print_string "Plus que 2"
- : unit = ()
```

SI ... ALORS ... SINON ...

```
if n > 2 then
          print_string "Plus que 2"
else
          print_string "Moins que 2";;
Moins que 2- : unit = ()
```

S'il y a plusieurs lignes à exécuter, on les délimite par begin ... end et on sépare les instructions d'un unique ;

```
if n > 2 then begin
          print_int n;
          print_string " est plus grand que 2"
end else begin
          print_int n;
          print_string " est plus petit que 2"
end;;

-3 est plus petit que 2- : unit = ()
```

Avec une fonction : $f(n) = \begin{cases} n+1 \text{ si } n > 0 \\ -n+4 \text{ sinon.} \end{cases}$

Avec une fonction : $f(n) = \begin{cases} n+1 \text{ si } n > 0 \\ -n+4 \text{ sinon.} \end{cases}$

```
# let f(n) = if n > 0 then n + 1 else -n + 4;;
val f : int -> int = <fun>
```

Avec une fonction : $f(n) = \begin{cases} n+1 \text{ si } n > 0 \\ -n+4 \text{ sinon.} \end{cases}$

```
# let f(n) = if n > 0 then n + 1 else -n + 4;;
val f : int -> int = <fun>
```

Remarque importante

Remarque qu'une fonction ne comporte pas de `return` : elle renvoie l'unique objet présent sur la pile d'exécution.

Vocabulaire

Tout objet informatique possède un **type**.

Limitation des entiers :

- dans l'intervalle $[-2^{30}; 2^{30} - 1] = [-1073741824; 1073741823]$ pour CAML 32 bits
- dans l'intervalle $[-2^{62}; 2^{62} - 1] \approx [-4,6.10^{18}; 4,6.10^{18}]$ pour CAML 64 bits

Différences avec Python

On peut connaitre le plus grand entier grâce à la constante `max_int`.

Ajouter / Soustraire

```
#2 + 3 - 1;;
- : int = 4
```

Multiplier

```
#7 * 4;;
- : int = 28
```

Vocabulaire

+, - et * sont **opérateurs binaires** (à deux arguments) **infixes** (un argument avant et un après le symbole).

Quotient (entier)

```
#7 / 4;;
- : int = 1
```

Reste

```
#7 mod 4;;
- : int = 3
```

Remarques :

- Le reste de $a \div b$ est défini comme l'unique entier r du même signe que a et tel que $|r| < |b|$.
- Il n'y a pas de fonction puissance ou racine carrée de prédéfinies sur les entiers

Différences avec Python

Vocabulaire

OCaml est un langage **fortement typé**,
cela signifie en particulier que :

- Les conversions implicites de types sont formellement interdites.
- Le typage d'une fonction est réalisé au moment de sa définition.
- La compilation peut détecter des erreurs de typage.

```
#let a = 3 and b = 1.2;;
a : int = 3
b : float = 1.2
#let c = a + b;;
Entrée interactive:
>let c = a + b;;
>            ^
Cette expression est de
type float,
mais est utilisée avec
le type int.
```

```
>>> a = 3
>>> b = 1.2
>>> c = a + b
>>> type(a)
<class 'int'>
>>> type(b)
<class 'float'>
>>> type(c)
<class 'float'>
```

- Opérations : +. ; -. ; *. et /.
- Fonctions primitives : **, sqrt, sin, float, ...
- Conversions :
  - Entier → flottant : `float_of_int 3` renvoie 3.0
  - Flottant → entier : `int_of_float 5.4` renvoie 5

- Ils peuvent prendre 2 valeurs :

```
#2 > 4;;
- : bool = false
#"maison" > "chateau";;
- : bool = true
```

Opérations sur les booléens :

- ET

```
# 1 < 2 && 1/0 = 1;;
Exception: Division_by_zero.
# 1 > 2 && 1/0 = 1;;
- : bool = false
```

- OU

```
# 1 < 2 || 1/0 = 1;;
- : bool = true
```

- NON

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

- OU exclusif n'existe pas...

Remarque : ces opérateurs sont à faible **précédence**, inutile donc de surcharger avec des parenthèses

Le type `unit` ne comporte qu'une seule valeur, notée ().

Les fonctions **à effet de bord** (ne rendent pas une valeur mais modifient une variable globale, impriment, ...) sont en général de type `... -> unit`.

Vocabulaire

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

```
#1, 1;;
- : int * int = 1, 1
#"pi", 3.14;;
- : string * float = "pi", 3.14
#(1,3);;
- : int * int = 1, 3
#1, (2, 3);;
- : int * (int * int) = 1, (2, 3)
```

- Récupérer la première composante d'un couple :

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

- Récupérer la seconde composante d'un couple :

```
#snd (4, 5);;
- : int = 5
```

- Pour un multiplet quelconque :

```
# let (a, _, b, _, _) = (4, 3, 6, 8, 9);;
val a : int = 4
val b : int = 6
```