Introduction au langage OCaml

Lycée Henri Poincaré - Nancy

27 janvier 2021

Première expression en OCaml

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

- Le `#` :
  - s'appelle un prompt
  - Remplace le `In[1]` de Spyder.
- `;;` à la fin du programme
- Réponse de OCaml : `type = valeur`

Sommaire

Sommaire

Types de base en OCaml

- Le type `unit` (`Nonetype` en Python) :

```
# ();;                    (* None en Python *)
- : unit = ()
# print_int 1;;  (* ou print_int(1) *)
1- : unit = ()
```

Types de base en OCaml

- Le type `unit` (`Nonetype` en Python) :

```
# ();;                (* None en Python *)
- : unit = ()
# print_int 1;;       (* ou print_int(1) *)
1- : unit = ()
```

- Le type `int` :

```
# (2*3)/4;;          (* division entiere *)
- : int = 1
```

  - Opérations : `+` `-` `*` `/` `mod`

Types de base en OCaml

- Le type `unit` (`Nonetype` en Python) :

```
# ();;                (* None en Python *)
- : unit = ()
# print_int 1;;       (* ou print_int(1) *)
1- : unit = ()
```

- Le type `int` :

```
# (2*3)/4;;          (* division entiere *)
- : int = 1
```

  - Opérations : `+` `-` `*` `/` `mod`
  - $\in [\![-2^{62}; 2^{62} - 1]\!]$

Types de base en OCaml

- Le type `unit` (`Nonetype` en Python) :

```
# ();;                (* None en Python *)
- : unit = ()
# print_int 1;;  (* ou print_int(1) *)
1- : unit = ()
```

- Le type `int` :

```
# (2*3)/4;;          (* division entiere *)
- : int = 1
```

  - Opérations : `+` `-` `*` `/` `mod`
  - $\in [\![-2^{62}; 2^{62} - 1]\!]$
  - Entiers signés sur 63 bits (pas sur 64 bits)

```
# 4611686018427387903;; (* = 2**62 - 1 *)
- : int = 4611686018427387903
# 4611686018427387903 + 1;;
- : int = -4611686018427387904
```

Types de base en OCaml

- Le type `float` :

```
# (2.*.3.)/.4.;;      (* division de flottants *)
- : float = 1.5
```

  - Opérations sur les `float` : `+.` `-.` `*.` `/.` `**` `sqrt` `cos` `log` ...
  - Flottants double précision (`binary64`)

Types de base en OCaml

- Le type `float` :

```
# (2.*.3.)/.4.;;      (* division de flottants *)
- : float = 1.5
```

  - Opérations sur les `float` : `+.` `-.` `*.` `/.` `**` `sqrt` `cos` `log` ...
  - Flottants double précision (`binary64`)

- Pas de conversion implicite en OCaml :

```
# 1 + 2.;; (* int + float *)
Characters 4-6:
  1 + 2.;; (* int + float *)
      ^^
Error: This expression has type float but an expression
       was expected of type int
```

Types de base en OCaml

- Le type `float` :

```
# (2.*.3.)/.4.;;      (* division de flottants *)
- : float = 1.5
```

  - Opérations sur les `float` : `+.` `-.` `*.` `/.` `**` `sqrt` `cos` `log` ...
  - Flottants double précision (`binary64`)

- Pas de conversion implicite en OCaml :

```
# 1 + 2.;; (* int + float *)
Characters 4-6:
  1 + 2.;; (* int + float *)
      ^^
Error: This expression has type float but an expression
       was expected of type int
```

- Conversions explicites :

```
# int_of_float 1.;;
- : int = 1
```

```
# float_of_int 1;;
- : float = 1.
```

Types de base en OCaml

- Le type `bool` :

```
1  # true;;  (* True en Python *)
2  - : bool = true
3  # false;; (* False en Python *)
4  - : bool = false
```

Python	OCaml	OCaml (à éviter)
< <= >= >	idem	
==	=	
!=	<>	
not bool	idem	
bool and bool	bool && bool	bool & bool
bool or bool	bool || bool	bool or bool

Types de base en OCaml

- Les tuples

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

```
# let t = 1,2,3;;
val t : int * int * int = (1, 2, 3)
# let a,b,c = t;;
val a : int = 1
val b : int = 2
val c : int = 3
```

Types de base en OCaml

- Les caractères :

```
# 'a';;                (* Caractères *)
- : char = 'a'
```

Types de base en OCaml

- Les caractères :

```
# 'a';;                (* Caractères *)
- : char = 'a'
```

- Chaînes de caractères :

```
# "Hello World !";; (* Chaines de caracteres *)
- : string = "Hello World !"
```

Types de base en OCaml

- Les caractères :

```
# 'a';;                (* Caractères *)
- : char = 'a'
```

- Chaînes de caractères :

```
# "Hello World !";; (* Chaines de caracteres *)
- : string = "Hello World !"
```

Attention : `string` de taille 1 $\neq$ `char` (contrairement à Python)

```
# "a" = 'b';;
Line 1, characters 6-9:
1 | "a" = 'b';;
           ^^^
Error: This expression has type char but an expression was
       expected of type string
```

Types de base en OCaml

- Messages d'erreurs :

```
# failwith "Message d'erreur";;
Exception: Failure "Message d'erreur".
```

Types de base en OCaml

- Messages d'erreurs :

```
# failwith "Message d'erreur";;
Exception: Failure "Message d'erreur".
```

Utilisés pour indiquer qu'on ne peut pas faire un calcul :

```
# failwith "Impossible de calculer la factorielle d'un
    nombre negatif";;
Exception: Failure "Impossible de calculer la
    factorielle d'un nombre negatif".
```

Sommaire

Variables en OCaml

Variables globales

- Syntaxe générale :

```
1  let <nom variable> = <valeur>;;
```

Variables en OCaml

Variables globales

- Syntaxe générale :

```
let <nom variable> = <valeur>;;
```

- Exemple :

```
# let a = 1;;          (* Affectation *)
val a : int = 1
# a + 1;;
- : int = 2
# a = 2;;              (* Test d'egalite *)
- : bool = false
```

Variables en OCaml

Variables locales

- Syntaxe générale :

```
1  let <nom variable> = <valeur> in
2      <expression>;;
```

Variables en OCaml

Variables locales

- Syntaxe générale :

```
let <nom variable> = <valeur> in
    <expression>;;
```

- Exemple :

```
# let x = 1 in
    x + 1;;
- : int = 2
# x;;
Characters 0-1:
  x;;
  ^
Error: Unbound value x
```

Sommaire

Expression conditionnelle

```
1  if <expression booleenne> then
2    <expression>
3  else
4    <expression>;;
```

Exercice

Étant donné une variable a, écrire une expression qui s'évalue en la valeur absolue de a.

Expression conditionnelle

```
1  if <expression booleenne> then
2    <expression>
3  else
4    <expression>;;
```

Exercice

Étant donné une variable a, écrire une expression qui s'évalue en la valeur absolue de a.

```
1  if a > 0 then
2    a
3  else
4    -a;;
```

Expression conditionnelle

```
1  if <expression booleenne> then
2    <expression>
3  else
4    <expression>;;
```

Exercice

Étant donné une variable `a`, écrire une expression qui s'évalue en la valeur absolue de `a`.

```
1  if a > 0 then
2    a
3  else
4    -a;;
```

Exercice

Écrire une expression qui affiche `"positif"` si `a` est positif, `"negatif"` si `a` est négatif, puis s'évalue en la valeur absolue de `a`.

Expression conditionnelle

- Solution de l'exercice :

```
1  if a > 0 then begin
2       print_string "positif";
3       a
4     end else begin
5       print_string "negatif";
6       -a
7     end;;
```

Expression conditionnelle

- Solution de l'exercice :

```
1  if a > 0 then begin
2       print_string "positif";
3       a
4     end else begin
5       print_string "negatif";
6       -a
7     end;;
```

- Syntaxe générale :

```
1  if <expression booleenne> then begin
2       <expression de type unit>;
3            ...
4       <expression de type unit>;
5       <expression>
6     end else begin
7       <expression de type unit>;
8            ...
9       <expression de type unit>;
10      <expression>
11    end;;
```

Expression conditionnelle

- Autre solution :

```
1  if a > 0 then (
2    print_string "positif";
3    a
4  ) else (
5    print_string "positif";
6    -a);;
```

Expression conditionnelle

- Autre solution :

```
1  if a > 0 then (
2    print_string "positif";
3    a
4  ) else (
5    print_string "positif";
6    -a);;
```

- **Ne fonctionne pas :**

```
1  if a > 0 then     (* Ne fonctionne pas *)
2    print_string "positif";
3    a
4  else
5    print_string "positif";
6    -a;;
7  4 | else
8        ^^^^
9  Error: Syntax error
```

Expression conditionnelle

Piège classique

Qu'affiche le programme suivant ?

```
1  if 1 = 0 then
2       print_int 1;
3       print_int 2;
4  print_int 3;;
```

Expression conditionnelle

Piège classique

Qu'affiche le programme suivant ?

```
1  if 1 = 0 then
2       print_int 1;
3       print_int 2;
4  print_int 3;;
```

- Réponse : 23
- La ligne 3 n'est pas dans le `if`
- L'indentation n'a pas signification en Ocaml ($\neq$ Python)

Expression conditionnelle

Piège classique

Qu'affiche le programme suivant ?

```
1  if 1 = 0 then
2       print_int 1;
3       print_int 2;
4  print_int 3;;
```

- Réponse : 23
- La ligne 3 n'est pas dans le `if`
- L'indentation n'a pas signification en Ocaml ($\neq$ Python)
- Pour afficher 3 :

Expression conditionnelle

Piège classique

Qu'affiche le programme suivant ?

```
1  if 1 = 0 then
2       print_int 1;
3       print_int 2;
4  print_int 3;;
```

- Réponse : 23
- La ligne 3 n'est pas dans le `if`
- L'indentation n'a pas signification en Ocaml ($\neq$ Python)
- Pour afficher 3 :

```
1  if 1 = 0 then begin
2       print_int 1;
3       print_int 2;
4     end;
5  print_int 3;
```

Sommaire

Type d'une fonction

- Exemple 1 : `int -> float` :
  - 1 argument de type `int`.
  - Renvoie un `float`.

Type d'une fonction

- Exemple 1 : `int -> float` :
  - 1 argument de type `int`.
  - Renvoie un `float`.

```
# float_of_int;;
- : int -> float = <fun>
# float_of_int 4;;
- : float = 4.
```

Type d'une fonction

- Exemple 1 : `int -> float` :
  - 1 argument de type `int`.
  - Renvoie un `float`.

```
# float_of_int;;
- : int -> float = <fun>
# float_of_int 4;;
- : float = 4.
```

- Exemple 2 : `int -> bool -> float` :

Type d'une fonction

- Exemple 1 : `int -> float` :
  - 1 argument de type `int`.
  - Renvoie un `float`.

```
# float_of_int;;
- : int -> float = <fun>
# float_of_int 4;;
- : float = 4.
```

- Exemple 2 : `int -> bool -> float` :
  - 1er argument de type `int`.
  - 2nd argument de type `bool`.
  - Renvoie un `float`.

Polymorphisme

Fonction polymorphe :

- Définition : plusieurs types autorisés pour l'un des arguments
- Le type de la fonction contient des `'a`, `'b`, `'c` ...

Polymorphisme

Fonction polymorphe :

- Définition : plusieurs types autorisés pour l'un des arguments
- Le type de la fonction contient des 'a, 'b, 'c ...

```
# min;;
- : 'a -> 'a -> 'a = <fun>
# min 2 5;;
- : int = 2
# min 2. 5.;;
- : float = 2.
# min 2 5.;;
Line 1, characters 6-8:
1 | min 2 5.;;
            ^^
Error: This expression has type float but an expression
    was expected of type int
```

Les fonctions non récursives

```
1  let <nom fonction> <argument(s)> =
2      <expression>;;
```

Les fonctions non récursives

```
1 let <nom fonction> <argument(s)> =
2     <expression>;;
```

Exercice 1

- Écrire une fonction `carre` qui prend en argument un entier `n` et renvoie son carré.
- Écrire une fonction `puiss_quatrieme` qui prend en argument un entier `n` et renvoie sa puissance quatrième.
- Donnez le type de vos fonctions.
- Testez lorsque `n` vaut `4`.

Les fonctions non récursives

```
let <nom fonction> <argument(s)> =
    <expression>;;
```

Exercice 1

- Écrire une fonction `carre` qui prend en argument un entier `n` et renvoie son carré.
- Écrire une fonction `puiss_quatrieme` qui prend en argument un entier `n` et renvoie sa puissance quatrième.
- Donnez le type de vos fonctions.
- Testez lorsque `n` vaut `4`.

Exercice 2

- Écrire une fonction `somme` qui prend en argument deux entiers `n` et `m`, et renvoie leur somme.
- Donnez le type de votre fonction.
- Testez lorsque `n` vaut `4` et `m` vaut `2`.

Correction de l'exercice 1

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

```
# carre 4;;
- : int = 16
```

```
# let puiss_quatrieme n =      (* 1ere solution *)
    let m = carre n in
    carre m;;
val puiss_quatrieme : int -> int = <fun>
```

```
# let puiss_quatrieme n =      (* 2eme solution *)
  carre (carre n);;
val puiss_quatrieme : int -> int = <fun>
```

```
# puiss_quatrieme 4;;
- : int = 256
```

Correction de l'exercice 2

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
```

```
# somme 4 2;;
- : int = 6
```

Attention : `int -> int -> int` $\neq$ `int * int -> int` :

```
# somme(4,2);;
Characters 5-10:
  somme(4,2);;
        ^^^^^
Error: This expression has type 'a * 'b
       but an expression was expected of type int
```

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
1  # let somme n m =
2        n + m;;
3  val somme : int -> int -> int = <fun>
```

  - `int -> int -> int` = `int -> (int -> int)`.

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
# somme 1 2;;
```

  - `int -> int -> int` = `int -> (int -> int)`.

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
# somme 1 2;;
- : int = 3
```

  - `int -> int -> int` = `int -> (int -> int)`.

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
# somme 1 2;;
- : int = 3
# somme 1;;
```

  - `int -> int -> int` = `int -> (int -> int)`.

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
# somme 1 2;;
- : int = 3
# somme 1;;
- : int -> int = <fun>
```

  - `int -> int -> int` = `int -> (int -> int)`.
  - `somme 1` est une fonction qui associe à `m` l'entier `1+m`.
  - `somme` est une fonction qui associe à `n` la fonction `somme n`.

La forme curryfiée

- Exemple de forme curryfiée (à préférer) :

```
# let somme n m =
    n + m;;
val somme : int -> int -> int = <fun>
# somme 1 2;;
- : int = 3
# somme 1;;
- : int -> int = <fun>
```

  - `int -> int -> int` = `int -> (int -> int)`.
  - `somme 1` est une fonction qui associe à `m` l'entier `1+m`.
  - `somme` est une fonction qui associe à `n` la fonction `somme n`.

- Exemple de forme non-curryfiée (à éviter) :

```
# let somme_bis (n,m) =
    n + m;;
val somme_bis : int * int -> int = <fun>
# somme_bis (1,2);;
- : int = 3
```

Type d'une fonction

- Exemple 3 : `(int -> bool) -> float` :

Type d'une fonction

- Exemple 3 : `(int -> bool) -> float` :
  - 1 argument de type `int -> bool` (cet argument est une fonction).
  - Renvoie un `float`.

Type d'une fonction

- Exemple 3 : `(int -> bool) -> float` :
  - 1 argument de type `int -> bool` (cet argument est une fonction).
  - Renvoie un `float`.

Exercice

Quel est le type de la fonction :

```
1  let apply f x = 2 * (f (x+1));;
```

Type d'une fonction

- Exemple 3 : `(int -> bool) -> float` :
  - 1 argument de type `int -> bool` (cet argument est une fonction).
  - Renvoie un `float`.

Exercice

Quel est le type de la fonction :

```
1  let apply f x = 2 * (f (x+1));;
```

Réponse : `(int -> int) -> int -> int`

- 1er argument de type `int -> int` (cet argument est une fonction).
- 2ème argument de type `int`
- Renvoie un `int`.

Sommaire

Filtrage

- Remplace les `if ... then ... else`
- Syntaxe générale :

```
match <expression 1> with
   | <pattern> -> <expression>
   ...
   | <pattern> -> <expression>
```

- Pattern $\approx$ décomposition de `<expression 1>`.

Filtrage

- Remplace les `if ... then ... else`
- Syntaxe générale :

```
match <expression 1> with
   | <pattern> -> <expression>
   ...
   | <pattern> -> <expression>
```

- Pattern $\approx$ décomposition de `<expression 1>`.

Exercice

Écrire une fonction `xor` qui calcule le ou exclusif de deux booléens :

```
# xor false false;;
- : bool = false
# xor false true;;
- : bool = true
```

```
# xor true false;;
- : bool = true
# xor true true;;
- : bool = false
```

Solution de l'exercice

```
let xor b1 b2 = match b1,b2 with
  | false,false -> false
  | false,true  -> true
  | true ,false -> true
  | true ,true  -> false;;
```

```
let xor b1 b2 = match b1,b2 with
  | false,_     -> b2      (* _ = valeur quelconque *)
  | _    ,true  -> false
  | _           -> true;;  (* Dans tous les autres cas *)
```

```
let xor b1 b2 = match b1 with
  | _ when b1=b2 -> false
  | _            -> true;;
```

```
let xor b1 b2 = b1 <> b2;;
```

Sommaire

Fonctions récursives

- Remplacent les boucles `for` et `while`

Fonctions récursives

- Remplacent les boucles `for` et `while`
- Définition : une fonction récursive est une fonction qui s'appelle elle même

Fonctions récursives

- Remplacent les boucles `for` et `while`
- Définition : une fonction récursive est une fonction qui s'appelle elle même
- Syntaxe générale :

```
1  let rec <nom fonction> <argument(s)> = (* Penser au rec *)
2      <expression>;;
```

Fonctions récursives

- Remplacent les boucles `for` et `while`
- Définition : une fonction récursive est une fonction qui s'appelle elle même
- Syntaxe générale :

```
1  let rec <nom fonction> <argument(s)> = (* Penser au rec *)
2      <expression>;;
```

Exercice

Écrire une fonction `fact: int -> int` qui calcule la factorielle d'un nombre.

Fonctions récursives

- Remplacent les boucles `for` et `while`
- Définition : une fonction récursive est une fonction qui s'appelle elle même
- Syntaxe générale :

```
let rec <nom fonction> <argument(s)> = (* Penser au rec *)
    <expression>;;
```

Exercice

Écrire une fonction `fact: int -> int` qui calcule la factorielle d'un nombre.

```
let rec fact n = match n with
  | _ when n < 0 -> failwith "Nombre negatif"
  | 0 -> 1
  | _ -> n * fact(n-1);;
```

Fonctions anonymes

- Maths : $x \mapsto x + 1$
- Python : `lambda x: x+1`
- Ocaml : `fun x -> x+1`

Fonctions anonymes

- Maths : $x \mapsto x + 1$
- Python : `lambda x: x+1`
- Ocaml : `fun x -> x+1`

Exercice

- Écrire une expression de type `('a -> 'b) -> 'a -> 'b` qui prend en argument une fonction f ainsi que x, et renvoie $f(x)$.
- Tester lorsque $f : x \mapsto e^x + 1$ et $x = 0$

Fonctions anonymes

- Maths : $x \mapsto x + 1$
- Python : `lambda x: x+1`
- Ocaml : `fun x -> x+1`

Exercice

- Écrire une expression de type `('a -> 'b) -> 'a -> 'b` qui prend en argument une fonction f ainsi que x, et renvoie $f(x)$.
- Tester lorsque $f : x \mapsto e^x + 1$ et $x = 0$

```
# fun f x -> f x;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

Fonctions anonymes

- Maths : $x \mapsto x + 1$
- Python : `lambda x: x+1`
- Ocaml : `fun x -> x+1`

Exercice

- Écrire une expression de type `('a -> 'b) -> 'a -> 'b` qui prend en argument une fonction f ainsi que x, et renvoie $f(x)$.
- Tester lorsque $f : x \mapsto e^x + 1$ et $x = 0$

```
# fun f x -> f x;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

```
# (fun f x -> f x) (fun x -> exp x +. 1.) 0.;;
- : float = 2.
```

Fonctions anonymes

Exercice

Définir une fonction `somme: int -> int -> int` de deux manières différentes.

Fonctions anonymes

Exercice

Définir une fonction `somme: int -> int -> int` de deux manières différentes.

```
1  let somme n m = n + m;;
2  let somme = fun n m -> n + m;;
```

Fonctions anonymes

- Deux syntaxes équivalentes :

```
1  fun x -> match x with
2      | ...
```

```
1  function
2      | ...
```

Fonctions anonymes

- Deux syntaxes équivalentes :

```
1  fun x -> match x with
2      | ...
```

```
1  function
2      | ...
```

Exercice

Écrire une fonction `f: int -> int` qui renvoie 1 si son entrée est positive ou nulle et -1 sinon.

Solution de l'exercice

- 1ère solution :

```
let f x = match x with
  | x when x >= 0 -> 1
  | _ -> -1;;
```

- 2ème solution :

```
let f = function
  | x when x >= 0 -> 1
  | _ -> -1;;
```

Exercice

Quel est le type et que fait la fonction suivante ?

```
let g x = function
  | a when a >= 0 -> x
  | _ -> -x;;
```

Solution de l'exercice

- Type : `int -> int -> int`
- `g` est équivalente à :

```
let g x = fun y -> match y with
  | a when a >= 0 -> x
  | _ -> -x;;
```

```
let g x y = match y with
  | a when a >= 0 -> x
  | _ -> -x;;
```

- `g x y` vaut $\begin{cases} \texttt{x} & \text{si } \texttt{y >= 0} \\ \texttt{-x} & \text{sinon} \end{cases}$

Sommaire

Programmation impérative en Ocaml

- Boucles `for` et `while` existent en Ocaml

Programmation impérative en Ocaml

- Boucles `for` et `while` existent en Ocaml
- Attention, une variable Ocaml ne peut pas être modifiée

Exercice

Qu'affiche le programme suivant ?

```
1  let a = 0 in
2      for i = 1 to 3 do
3        print_int a; print_char ' ';
4        let a = a + i in
5        print_int a; print_char ' ';
6      done;
7      print_int a; print_newline();;
```

Programmation impérative en Ocaml

- Boucles `for` et `while` existent en Ocaml
- Attention, une variable Ocaml ne peut pas être modifiée

Exercice

Qu'affiche le programme suivant ?

```
1  let a = 0 in
2      for i = 1 to 3 do
3        print_int a; print_char ' ';
4        let a = a + i in
5        print_int a; print_char ' ';
6      done;
7      print_int a; print_newline();;
```

- Réponse : `0 1 0 2 0 3 0`

Programmation impérative en Ocaml

- Boucles `for` et `while` existent en Ocaml
- Attention, une variable Ocaml ne peut pas être modifiée

Exercice

Qu'affiche le programme suivant ?

```
1  let a = 0 in
2      for i = 1 to 3 do
3        print_int a; print_char ' ';
4        let a = a + i in
5        print_int a; print_char ' ';
6      done;
7      print_int a; print_newline();;
```

- Réponse : `0 1 0 2 0 3 0`
- Explication : l'instruction `let a = a + i in` :
  - Créé une variable locale (qui existe pendant un tour de boucle)
  - Ne modifie pas la variable créée par `let a = 0 in`

Références

- Référence $\approx$ variable modifiable

Références

- Référence ≈ variable modifiable
- Déclaration :

```
# let a = ref 0;;
val a : int ref = {contents = 0}
# let b = ref 3.1415;;
val b : float ref = {contents = 3.1415}
# let c = ref 'c';;
val c : char ref = {contents = 'c'}
```

Références

- Référence ≈ variable modifiable
- Déclaration :

```
# let a = ref 0;;
val a : int ref = {contents = 0}
# let b = ref 3.1415;;
val b : float ref = {contents = 3.1415}
# let c = ref 'c';;
val c : char ref = {contents = 'c'}
```

- Accès :

```
# !a;;
- : int = 0
# !b;;
- : float = 3.1415
# !c;;
- : char = 'c'
```

- Modification :

```
# a := 4;
a;;
- : int ref = {contents = 4}
# b := 2. *. !b;
b;;
- : float ref = {contents = 6.283}
# c := char_of_int(int_of_char !c + 1);
c;;
- : char ref = {contents = 'd'}
```

Programmation impérative en Ocaml

- Modification :

```
# a := 4;
a;;
- : int ref = {contents = 4}
# b := 2. *. !b;
b;;
- : float ref = {contents = 6.283}
# c := char_of_int(int_of_char !c + 1);
c;;
- : char ref = {contents = 'd'}
```

Exercice

- Définir une référence `a` contenant 3
- Mettre le contenu de `a` au carré
- Afficher le contenu de `a`

Solution de l'exercice

```
# let a = ref 3 in
    a := !a * !a;
    print_int !a;
    print_newline();;
9
- : unit = ()
```

Boucles `for`

```
for <nom variable> = <valeur ini> to <valeur finale> do
  <expr de type unit>;
done;;
```

Remarque : valeur initiale/finale incluse

Boucles `for`

```
1  for <nom variable> = <valeur ini> to <valeur finale> do
2    <expr de type unit>;
3  done;;
```

Remarque : valeur initiale/finale incluse

Exercice

Calculer la somme des entiers de 1 à 9

Boucles `for`

```
for <nom variable> = <valeur ini> to <valeur finale> do
  <expr de type unit>;
done;;
```

Remarque : valeur initiale/finale incluse

Exercice

Calculer la somme des entiers de 1 à 9

```
let somme = ref 0 in
    for i = 1 to 9 do
      somme := !somme + i;
    done;
    !somme;;
```

Boucles `while`

```
1  while <expr booleenne> do
2    <expr de type unit>;
3  done;;
```

Programmation impérative en Ocaml

Boucles `while`

```
while <expr booleenne> do
  <expr de type unit>;
done;;
```

Exercice

Calculer la somme des entiers de 1 à 9

Boucles `while`

```
while <expr booleenne> do
  <expr de type unit>;
done;;
```

Exercice

Calculer la somme des entiers de 1 à 9

```
let somme = ref 0
and i = ref 0 in
    while !i < 10 do
      somme := !somme + !i;
      incr(i);    (* i := !i + 1 *)
    done;
    !somme;;
```

Sommaire

Propriétés du langage OCaml

- Langage fortement typé :
  - Type pour chaque objet (variable, fonction, . . .)
  - Pas de conversions implicites

Propriétés du langage OCaml

- Langage fortement typé :
  - Type pour chaque objet (variable, fonction, . . .)
  - Pas de conversions implicites
- Indentation pas obligatoire :
  - **À faire :** même indentation qu'en Python
  - **Interdit :** on pourrait tout écrire sur une ligne

Langages fonctionnels & Langages impératifs

- Les expressions :
  - Définition : énoncé qui s'évalue en une valeur
  - Exemple : `1+1` est de type `int` et s'évalue en `2`

Langages fonctionnels & Langages impératifs

- Les expressions :
  - Définition : énoncé qui s'évalue en une valeur
  - Exemple : `1+1` est de type `int` et s'évalue en `2`
- Les instructions :
  - Définition : ordre donné à l'ordinateur
  - Exemple : `print_int 2;;`

Langages fonctionnels & Langages impératifs

- Langages impératifs :
  - Utilisation de boucles `for` et `while`
  - Programme = suite d'instructions

```
1  let factorielle n =
2    let res = ref 1 in
3    for i = 2 to n do
4      res := !res * i
5    done;
6    !res;;
```

Langages fonctionnels & Langages impératifs

- Langages impératifs :
  - Utilisation de boucles `for` et `while`
  - Programme = suite d'instructions

```
1  let factorielle n =
2    let res = ref 1 in
3    for i = 2 to n do
4      res := !res * i
5    done;
6    !res;;
```

- Langages fonctionnels (ex : Ocaml) :
  - Utilisation de fonctions récursives
  - Programme = évaluation d'une expression (souvent récursive)

```
1  let rec factorielle n = match n with
2    | n when n < 0 -> failwith "n < 0"
3    | 0 -> 1
4    | n -> n * factorielle (n-1);;
```

Sommaire

Forcer le typage

Conseil : forcez le type de vos fonctions ($\rightsquigarrow$ messages d'erreurs plus simples).

Exemple

```
# let somme (n: int) (m: int): int =
    n + m;;
val somme : int -> int -> int = <fun>
```

Forcer le typage

Conseil : forcez le type de vos fonctions ($\rightsquigarrow$ messages d'erreurs plus simples).

Exemple

```
# let somme (n: int) (m: int): int =
    n + m;;
val somme : int -> int -> int = <fun>
```

Exercice

Quel est le type de la fonction suivante ?

```
let xor b1 b2 = b1 <> b2;;
```

Comment obtenir une fonction de type `bool -> bool -> bool` ?

Solution de l'exercice

```
# let xor b1 b2 = b1 <> b2;;
val xor : 'a -> 'a -> bool = <fun>
```

```
# let xor (b1: bool) (b2: bool): bool = b1 <> b2;;
val xor : bool -> bool -> bool = <fun>
```

Solution de l'exercice

```
# let xor b1 b2 = b1 <> b2;;
val xor : 'a -> 'a -> bool = <fun>
```

```
# let xor (b1: bool) (b2: bool): bool = b1 <> b2;;
val xor : bool -> bool -> bool = <fun>
```

Exercice

Forcez le typage de la fonction :

```
let f = function
  | 0 -> true
  | _ -> false;;
```

Solution de l'exercice

```
# let xor b1 b2 = b1 <> b2;;
val xor : 'a -> 'a -> bool = <fun>
```

```
# let xor (b1: bool) (b2: bool): bool = b1 <> b2;;
val xor : bool -> bool -> bool = <fun>
```

Exercice

Forcez le typage de la fonction :

```
let f = function
  | 0 -> true
  | _ -> false;;
```

```
let f: int -> bool = function
  | 0 -> true
  | _ -> false;;
```

Comment typer une expression ?

- Pour typer une expression, Caml utilise un algorithme récursif

Exercice

Donner le type des fonctions suivantes :

```
# let f g h x = (g (h x)) + (g x);;
```

```
# let f g h = (g h) + (h g);;
```

Comment typer une expression ?

- Pour typer une expression, Caml utilise un algorithme récursif

Exercice

Donner le type des fonctions suivantes :

```
# let f g h x = (g (h x)) + (g x);;
val f : ('a -> int) -> ('a -> 'a) -> 'a -> int = <fun>
```

```
# let f g h = (g h) + (h g);;
```

Comment typer une expression ?

- Pour typer une expression, Caml utilise un algorithme récursif

Exercice

Donner le type des fonctions suivantes :

```
# let f g h x = (g (h x)) + (g x);;
val f : ('a -> int) -> ('a -> 'a) -> 'a -> int = <fun>
```

```
# let f g h = (g h) + (h g);;
Line 1, characters 23-24:
1 | let f g h = (g h) + (h g);;
                           ^
Error: This expression has type ('a -> 'b) -> int
       but an expression was expected of type 'a
       The type variable 'a occurs inside ('a -> 'b) ->
    int
```

Solutions des exercices

```
1  # let f g h x = (g (h x)) + (g x);;
```

- Pour que l'expression `... + (g x)` soit bien typée, on doit avoir :

$$\begin{cases} \texttt{g: 'a -> int} \\ \texttt{x: 'a} \end{cases}$$

- Pour que l'expression `(g (h x)) + ...` soit bien typée, on doit avoir :

$$\{\texttt{h: 'a -> 'a}$$

Solutions des exercices

```
1  # let f g h = (g h) + (h g);;
```

- Pour que l'expression `(g h) + ...` soit bien typée, on doit avoir :

$$\begin{cases} \texttt{g: 'a -> int} \\ \texttt{h: 'a} \end{cases}$$

- Pour que l'expression `... + (h g)` soit bien typée, on doit avoir :

```
h: ('a -> int) -> int
```

- D'après le type de `h`, on a `'a = ('a -> int) -> int`.
  $\rightsquigarrow$ Impossible car `'a` apparaît des deux côtés de l'équation.