OCaml

September 11, 2021

1 Variables

1.1 Définition d'une variable

Une variable possède 3 propriétés:
- un **nom** (exemple : `x`)
- une **valeur** (exemple : 42)
- un **type** (exemple : entier)

En OCaml, on définit une variable de la façon suivante :

```
let variable = valeur
```

Par exemple, pour définir une variable `x` valant 42 :

[1]:
```
let x = 42
```

[1]:
```
val x : int = 42
```

OCaml nous répond que `x` a pour valeur 42 et est de type `int` (*integer*, c'est à dire entier). La variable `x` est ici définie globalement, c'est à dire accessible partout dans ce notebook.

On peut alors faire des calculs avec la valeur de `x` :

[2]:
```
3*x - 2 (* OCaml remplace x par 42 et fait le calcul *)
```

[2]:
```
- : int = 124
```

Il est possible d'avoir une expression (un calcul) à droite d'une définition :

[3]:
```
let a = 1 + 2 + 3 + 4 (* la valeur à droite de = est calculée puis mise dans a␣
↪*)
```

[3]:
```
val a : int = 10
```

Exercice : définissez une variable a égale à 752. Puis définissez une variable b égale à $54a$.
Rappel : vous pouvez appuyer sur la touche B pour faire apparaître une case de code et répondre dedans. Ici vous pouvez utiliser deux cases, ou séparer les deux instructions avec `;;`.

[4]:
```
let a = 752;;
let b = 54*a;;
```

[4]:
```
val a : int = 752
```

[4]:
```
val b : int = 40608
```

1.2 Variables locales

Il est possible de définir une **variable locale** en utilisant la forme `let y = ... in ...`
y existe est alors seulement dans le in (...)

[5]:
```
let y = -1 in y (* y est accessible dans le in *)
```

[5]:
```
- : int = -1
```

[6]:
```
y (* utiliser y ici donne une erreur  *)
```

```
File "[6]", line 1, characters 0-1:
1 | y (* utiliser y ici donne une erreur  *)
    ^
Error: Unbound value y
```

Exercice : en utilisant `let ... in ...`, définissez une variable a égale à 752, puis définissez une variable b égale à $54a$.

[7]:
```
let a = 752 in
let b = 54*a in
b
```

[7]:
```
- : int = 40608
```

Remarque Il est possible d'utiliser des parenthèses ou `begin ... end` pour délimiter le in :

```
let a = 3 in
(
...
)
```

1.3 Opérations numériques

Nous avons déjà vu l'addition et la soustraction de 2 entiers. Il est aussi possible de multiplier :

[8]:
```
3 * 14
```

[8]:
```
- : int = 42
```

On peut effectuer la division **entière** (ou encore : quotient de la division euclidienne) de 2 entiers :

[9]:
```
3 / 2 (* division entière *)
```

[9]:
```
- : int = 1
```

La division entière de x par y est, par définition, la partie entière de $\frac{x}{y}$.
Dans l'exemple ci-dessus, `3 / 2` est donc la partie entière de $\frac{3}{2} = 1.5$, c'est à dire 1.

Attention : la division entière `3 / 2` en OCaml correspond à `3 // 2` en Python.

En plus des entiers (`int`), OCaml permet de définir des nombres à virgules (`float`, pour flottant) :

[10]:
```
let pi = 3.141592
```

[10]:
```
val pi : float = 3.141592
```

Les opérateurs d'addition, soustraction, multiplication, division doivent s'utiliser avec un `.` (point). Par exemple :

[11]:
```
pi +. 2.618 (* noter le . après + *)
```

[11]:
```
- : float = 5.759592
```

Attention : c'est le point (.) et non pas la virgule qui est utilisé pour les flottants.

Il est possible de calculer x^y, où x et y sont des **flottants** avec `**` :

[12]:
```
2.718 ** 3.14 (* x puissance y *)
```

[12]:
```
- : float = 23.0963461891915607
```

Il n'est pas possible d'utiliser `**` sur des `int`. Par contre on peut utiliser 4.0 ou 4. au lieu de 4 pour avoir des flottants et utiliser `**` :

[13]:
```
2. ** 10.
```

[13]:
```
- : float = 1024.
```

Exercice 1. Stocker la valeur 42^2 dans une variable a, en utilisant `*`. 2. En déduire la valeur de 42^4. 3. Calculer la valeur de 2^{10} en utilisant le moins de multiplications possibles.

[14]:
```
(* 1. *)
let a = 42*42;;


(* 2. *)
a*a;;

(* 3. *)
let deux_puiss_2 = 2*2 in
let deux_puiss_4 = deux_puiss_2*deux_puiss_2 in
let deux_puiss_8 = deux_puiss_4*deux_puiss_4 in
deux_puiss_8 * deux_puiss_2;; (* calcul de 2 puissance 10 en 4 multiplications␣
↪*)
```

[14]:
```
val a : int = 1764
```

Exercice

1. Stocker dans 3 variables a, b, c les valeurs 2, 5 et 3. On pourra utiliser let a, b, c = …, …, … pour définir 3 variable simultanément.
2. Stocker dans une variable *delta* le discriminant de l'équation $ax^2 + bx + c = 0$.
3. Calculer toutes les solutions de l'équation précédente.

[15]:
```
(* 1. *)
let a, b, c = 2., 6., 3.;;
```

[15]:
```
val a : float = 2.
val b : float = 6.
val c : float = 3.
```

[16]:
```
(* 2. *)
let delta = b**2. -. 4.*.a*.c;;
```

[16]:
```
val delta : float = 12.
```

[17]:
```
(* 3. Le discriminant étant positif, il y a deux solutions égales à : *)
(-.b -. sqrt delta)/.(2.*.a);;
(-.b +. sqrt delta)/.(2.*.a);;
```

[17]:
```
- : float = -2.36602540378443837
```

[17]:
```
- : float = -0.633974596215561403
```

Une autre opération importante est le **modulo** (ou : reste de la division euclidienne) de 2 entiers `a` et `b`, notée `a mod b` en OCaml. Mathématiquement, il s'agit de l'entier r vérifiant :

$$a = bq + r$$

$$0 \leq r < b$$

(q est le quotient, égal à `a / b` en OCaml)
Cette opération `mod` sert notamment à tester la divisibilité : `a` est divisible par `b` si et seulement si `a mod b = 0`.

Exercice : Est-ce que 527 est divisible par 17?

[18]:
```
527 mod 17 (* c'est égal à 0 donc 17 divise bien 527 *)
```

[18]:
```
- : int = 0
```

1.4 Overflow

L'ordinateur stocke toutes les variables dans la mémoire RAM de l'ordinateur, en binaire (suite de 0 et de 1). Comme la mémoire RAM sur un ordinateur n'est pas infinie, on ne peut pas stocker des nombres de taille arbitraire.
`max_int` est le plus grand entier que l'on peut stocker dans une variable :

[19]:
```
max_int
```

[19]:
```
- : int = 4611686018427387903
```

Si on dépasse cet entier (ce qu'on appelle **integer overflow**), on tombe sur le plus petit entier représentable :

[20]:
```
max_int + 1
```

[20]:
```
- : int = -4611686018427387904
```

Le dépassement d'entier est une source fréquente de bug, qui a causé par exemple le crash de la fusée Ariane 5.

Les float sont également limités :

[21]:
```
max_float
```

[21]:
```
- : float = 1.79769313486231571e+308
```

Les flottants sont codées suivant la norme IEEE754 sous la forme scientifique en utilisant 64 bits (sur un processeur 64 bits) dont 52 bits pour les chiffres après la virgule.
La précision des flottants est donc limitée : par exemple, on ne peut pas stocker $\sqrt{2}$ ou π de façon

exacte sur l'ordinateur puisqu'il s'agit de nombres irrationnels (c'est à dire avec une infinité de décimales, qui ne se répètent pas).

[22]:
```
2.**0.5 (* seulement une partie des décimales est stockée *)
```

[22]:
```
- : float = 1.41421356237309515
```

[23]:
```
(2.**0.5)**2.0  (* ne donne pas 2.0 à cause des erreurs d'arrondis *)
```

[23]:
```
- : float = 2.00000000000000044
```

Le calcul de $\sqrt{2}$ par OCaml s'arrête à la 16ème décimale (ce qui correspond à 52 bits après la virgules, puisque 2^{-52} est du même ordre de grandeur que 10^{-16}). Ce nombre 2^{-52} est appelé **epsilon machine**.

Autre exemple :

[24]:
```
0.1 +. 0.2  (* ne donne pas 0.3 *)
```

[24]:
```
- : float = 0.300000000000000044
```

`0.1` n'étant pas représentable de façon exacte en base 2, il est tronqué et engendre des erreurs d'arrondis.

La plus grande valeur de l'exposant est 1023 :

[25]:
```
2.**1023.
```

[25]:
```
- : float = 8.98846567431158e+307
```

[26]:
```
2.**1024.
```

[26]:
```
- : float = infinity
```

1.5 Unit

Il existe une valeur spéciale `()` qui signifie "rien" (un peu comme le None de Python). Le type de `()` est `unit`. Par exemple, afficher un entier avec `print_int` fait un effet de bord (affichage sur l'écran) mais ne renvoie pas de valeur :

[27]:
```
print_int 42
```

[27]:
```
- : unit = ()
```

Pour que le résultat s'affiche, il faut revenir à la ligne avec `print_newline` :

[28]:
```
print_newline ()
```

[28]:
```
- : unit = ()
```

On verra dans un prochain cours le fonctionnement plus détaillé de `print_newline`.

Il aurait été possible de combiner ces deux instructions avec `;`, qui permet d'exécuter consécutivement plusieurs instructions :

[29]:
```
print_int 31;
print_newline ()
```

[29]:
```
- : unit = ()
```

`;;` est similaire à `;`, mais permet de séparer complètement plusieurs instructions, ce qui signifie que les variables définies avec `in` ne sont plus accessibles :

[30]:
```
let a = 3 in
print_int a;
a (* a reste accessible : il est toujours dans le in *)
```

[30]:
```
- : int = 3
```

[31]:
```
let a = 3 in
print_int a;;
a;; (* a n'est pas accessible ici *)
```

[31]:
```
- : unit = ()
```

[31]:
```
- : float = 2.
```

1.6 Présentation du code

Contrairement à Python, l'indentation du code n'a pas d'importance en OCaml. En effet les espaces et sauts de lignes sont ignorés :

[32]:
```
let a
  =
      6 (* indentation n'importe comment *)
```

[32]:
```
val a : int = 6
```

On veillera toutefois à écrire du code aussi clair et lisible que possible.

1.7 Références

Contrairement à ce que son nom l'indique, une variable définie comme on l'a fait jusqu'à maintenant est en fait… **constante** :

[33]:
```
let x = 3;;
x = 4;; (* ceci ne MODIFIE pas x, mais teste si la valeur de x est 4 *)
x;; (* x vaut toujours 3 : il ne peut pas être modifié *)
```

[33]:
```
val x : int = 3
```

Pour pouvoir modifier une variable, on peut utiliser une **référence**. Pour définir une référence on utilise `ref` :

```
let variable = ref valeur (* définition d'une référence *)
```

Pour modifier une référence on utilise `:=` :

```
variable := valeur (* modification d'une référence *)
```

Pour obtenir la valeur d'une référence on utilise `!` :

```
!variable (* donne la valeur d'une référence *)
```

[34]:
```
let x = ref 3;; (* définie une variable x qui "pointe" vers un emplacement␣
↪mémoire contenant 3 *)
x := 4;; (* modifie la valeur pointé par x avec := *)
!x;; (* la valeur a bien été modifiée *)
```

[34]:
```
val x : int ref = {contents = 3}
```

On remarque que l'instruction `x := 4` a pour valeur de retour `()` : c'est un effet de bord qui modifie `x` mais ne renvoie pas de résultat.

Une référence `a` stocke en fait une adresse mémoire. La valeur de `a` est stockée dans cette adresse mémoire (et c'est cette valeur qui est modifiée avec `a := ...`).
Remarque : on reverra cette idée avec les pointeurs en C.

Exercice : Que valent `!x` et `!y` après avoir exécuté le bout de code suivant? À quoi sert ce code?

```
let x = ref 5;;
let y = ref 27;;
x := !x + !y;
```

```
y := !x - !y;
x := !x - !y
```

[35]:
```
let x = ref 5 in
let y = ref 27 in
x := !x + !y;
y := !x - !y;
x := !x - !y;
!x, !y;; (* x vaut 27, y vaut 5 : ce bout de code échange les valeurs de x et y␣
 ↪*)
```

[35]:
```
- : int * int = (27, 5)
```

Remarque : utiliser des `ref` rend souvent le code plus compliqué, on n'en utilisera que lorsque c'est nécessaire.

2 Fonctions

2.1 Utiliser une fonction

OCaml est un langage fonctionnel, ce qui signifie que : - les fonctions y occupent une place importante et peuvent être manipulées un peu comme des variables - les fonctions sont censées ne pas effectuer d'effet de bord, c'est à dire d'action sur l'extérieur de la fonction (pas de modification de variable globale, pas d'écriture dans un fichier…)

Pour utiliser une fonction `f` sur une valeur `x`, on écrira simplement `f x` (et non pas `f(x)`).

Un certain nombre de fonctions sont déjà définies en OCaml. Par exemple, la racine carrée :

[1]:
```
sqrt 2.0 (* renvoie une approximation de racine de 2 *)
```

[1]:
```
- : float = 1.41421356237309515
```

Chaque fonction possède une **signature**, qui donne les types des paramètres (valeurs en entrée de la fonction) et le type de la valeur de retour.

[2]:
```
sqrt
```

[2]:
```
- : float -> float = <fun>
```

`float -> float` signifie que `sqrt` est une fonction qui prend un flottant en entrée et renvoie un flottant. On ne peut donc pas l'appliquer sur un entier :

[3]:
```
sqrt 2 (* erreur : on donne un entier à sqrt qui attend un flottant *)
```

```
File "[3]", line 1, characters 5-6:
1 | sqrt 2 (* erreur : on donne un entier à sqrt qui attend un flottant *)
```

```
          ^
Error: This expression has type int but an expression was expected of type
         float
```

2.2 Définir une fonction

En OCaml, une fonction se définie de la façon suivante :

```
let nom_fonction nom_argument = ...
```

où `...` est le corps de la fonction, c'est à dire ce qui est exécuté lorsqu'on utilise la fonction.

La valeur renvoyée par la fonction est celle de la dernière instruction (pas besoin de `return`).

Définissons par exemple la fonction $f : x \longmapsto 2x$:

[4]:
```
let f x = 2*x
```

[4]:
```
val f : int -> int = <fun>
```

OCaml nous dit que `f` est de type `int -> int`, ce qui signifie que `f` prend un entier en entrée et renvoie un entier en sortie. Ceci est similaire à la notation mathématique $f : \mathbb{N} \longrightarrow \mathbb{N}$.

On peut ensuite utiliser `f` et récupérer la valeur de retour :

[5]:
```
f 3
```

[5]:
```
- : int = 6
```

Notons que `x` est une variable **muette** : elle n'existe qu'à l'intérieur de `f`, n'a aucun rapport avec une variable `x` définie précédemment et la fonction suivante définit exactement la même fonction :

[6]:
```
let f y = 2*y (* peu importe le nom de la variable muette y *)
```

[6]:
```
val f : int -> int = <fun>
```

Maintenant que `f` est définie, on peut calculer $f(3)$:

[7]:
```
f 3
```

[7]:
```
- : int = 6
```

Exercice : définir la fonction $f : x \longmapsto \frac{1}{\sqrt{1+x^2}}$ en OCaml.

[8]:
```
let f x = 1./.(1. +. x**2.)**0.5
```

[8]:
```
val f : float -> float = <fun>
```

Comme pour les variables, il est possible d'utiliser `in` pour spécifier la portée d'une fonction g

[9]:
```
let g x = x + 1 in
g 0  (* g est utilisable seulement dans le in *)
```

[9]:
```
- : int = 1
```

Exercice Donner la valeur de l'expression suivante :

```
let h x = f x + 1 in
h 3
```

[10]:
```
let h x = f x + 1 in (* x est remplacé par 3, f x est remplacé par 6 *)
h 3
```

```
File "[10]", line 1, characters 10-13:
1 | let h x = f x + 1 in (* x est remplacé par 3, f x est remplacé par 6 *)
              ^^^
Error: This expression has type float but an expression was expected of type
         int
```

2.3 Fonctions anonymes

Quand on a besoin d'utiliser une fonction une seule fois, on peut définir une fonction anonyme (sans nom) avec `fun`. C'est l'équivalent de `lambda` en Python.

[11]:
```
fun x -> x*2 (* définition d'une fonction anonyme *)
```

[11]:
```
- : int -> int = <fun>
```

[12]:
```
(fun x -> x*2) 3 (* applique une fonction anonyme sur la valeur 3 *)
```

[12]:
```
- : int = 6
```

Remarque : les deux définitions suivantes sont en fait complètement équivalentes.

```
let f x = ...
```

```
let f = fun x -> ...
```

Par exemple, on peut définir la fonction $f : x \longmapsto 2\sqrt{x}$ comme ceci :

[13]:
```
let f = fun x -> 2.0*.x**0.5
```

[13]:
```
val f : float -> float = <fun>
```

Remarque : On peut aussi définir une fonction avec `function x -> ...` mais `fun` est légèrement plus simple d'utilisation.

2.4 Fonctions de plusieurs variables

Il est possible de définir des fonctions avec plusieurs paramètres, par exemple :

[14]:
```
let sum x y = x + y
```

[14]:
```
val sum : int -> int -> int = <fun>
```

[15]:
```
sum 3 4 (* renvoie 3 + 4 *)
```

[15]:
```
- : int = 7
```

Le type de `sum` est `int -> int -> int`, ce qui peut paraître étrange. C'est équivalent à `int -> (int -> int)`, ce qui signifie que `sum` prend en entier en argument et renvoie une valeur de type `int -> int` (c'est à dire une fonction).
En effet :

[16]:
```
sum 3
```

[16]:
```
- : int -> int = <fun>
```

`sum 3` est une fonction qui prend en argument un entier `y` et qui renvoie `3 + y`, ce qu'on peut vérifier :

[17]:
```
let f = sum 3 in (* f est une fonction *)
f 4 (* renvoie sum 3 4, c'est à dire 7 *)
```

[17]:
```
- : int = 7
```

En fait, OCaml transforme automatiquement une fonction de plusieurs variables en une suite de fonctions à une variable (c'est ce qu'on appelle la **curryfication**) :

[18]:
```
let sum = fun x -> (fun y -> x + y) (* OCaml transforme la définition de sum
↪ci-dessus en celle-ci *)
```

[18]:
```
val sum : int -> int -> int = <fun>
```

[19]:
```
(sum 2) 3  (* le calcul effectué par OCaml lorsqu'on écrit sum 2 3 *)
```

[19]:
```
- : int = 5
```

La possibilité d'appliquer une fonction seulement sur certains arguments s'appelle l'**application partielle** de fonction. C'est un des avantages d'OCaml par rapport à Python.

De la même façon, une fonction OCaml à 3 arguments sera de type `... -> ... -> ... -> ....`

Exercice : Écrire une fonction `delta : float -> float -> float -> float` telle que `delta a b c` renvoie le discriminant de l'équation $ax^2 + bx + c = 0$.

[20]:
```
let delta a b c = b**2. -. 4.*.a*.c
```

[20]:
```
val delta : float -> float -> float -> float = <fun>
```

Une fonction peut aussi avoir aucune valeur en entrée. Dans ce cas, on lui donne l'argument `()` (de type unit). C'est le cas par exemple de `print_newline`, qui saute une ligne :

[21]:
```
print_int 0;
print_newline ();
print_int 1;
print_newline ();
```

```
0
1
```

[21]:
```
- : unit = ()
```

2.5 Polymorphisme

Reprenons notre 1er exemple de fonction :

[22]:
```
let f x = 2*x
```

[22]:
```
val f : int -> int = <fun>
```

OCaml sait que l'argument `x` de `f` est un `int` car on utilise l'opérateur `*` qui ne s'utilise que sur des entiers. Mais dans certaines fonctions, il n'y a pas de contrainte de type :

[23]:
```
let id x = x
```

[23]:
```
val id : 'a -> 'a = <fun>
```

Cette fonction `id` (pour identité) renvoie son argument sans le modifier. Comme aucune opération n'est appliquée sur `x`, il n'y a pas de contrainte sur son type. OCaml utilise alors `'a` pour désigner le type quelconque de `x`.
Notons que le type de retour de `id` est `'a` également : OCaml nous dit que `id` renvoie une valeur du même type que l'argument.

Exercice : donner le type des fonctions suivantes

```
let f x = 42
```

```
let f x y = y
```

```
let g x y f = x + f y
```

[24]:
```
let f x = 42;;
(* x est quelconque ('a) et le type de retour est int *)
(* donc f est de type 'a -> int *)
```

[24]:
```
val f : 'a -> int = <fun>
```

[25]:
```
let f x y = y;;
(* x est quelconque ('a), y aussi ('b) mais le type de retour est le même que␣
 ↪y ('b) *)
(* donc f est de type 'a -> 'b -> 'b *)
```

[25]:
```
val f : 'a -> 'b -> 'b = <fun>
```

[26]:
```
let g x y f = x + f y;;
(* x est un int, à cause du + *)
(* y est quelconque ('a) *)
(* f est une fonction qui prend un y (de type 'a) et renvoie un int, à cause du␣
 ↪+ *)
(* donc f est de type int -> 'a -> ()*)
```

[26]:
```
val g : int -> 'a -> ('a -> int) -> int = <fun>
```

2.6 Fonction comme argument

Il est possible d'utiliser une fonction en argument d'une autre fonction. Par exemple, la fonction suivante évalue une autre fonction en la valeur 0 :

[27]:
```
let eval f =
f 0
```

```
[27]: val eval : (int -> 'a) -> 'a = <fun>
```

```
[28]: let f x = 3*x + 7 in
      eval f
```

```
[28]: - : int = 7
```

Exercice : 1. On définit une fonction `h` :

```
let h f g x = f (g x)
```

Donner la valeur de l'expression :

```
h (fun x -> x*x) (fun x -> x + 1) 3
```

2. Donner le type de `h`.
3. À quoi sert `h`? Comment cette opération s'appelle-t-elle mathématiquement?

```
[29]: (* 1. et 2. *)
      let h f g x = f (g x);;
      h (fun x -> x*x) (fun x -> x + 1) 3;;
```

```
[29]: val h : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
[29]: - : int = 16
```

3. `h` compose deux fonctions `f` et `g` : `h f g` est une fonction équivalent à $f \circ g$.

2.7 Variable locale à une fonction

Il est possible de définir une variable dans une fonction :

```
[30]: let pow4 x = (* je saute une ligne ici pour plus de lisibilité *)
          let y = x*x in  (* y est utilisable seulement dans pow4 *)
          y*y (* renvoie x puissance 4 *)
```

```
[30]: val pow4 : int -> int = <fun>
```

```
[31]: pow4 2 (* test de notre fonction *)
```

```
[31]: - : int = 16
```

On peut aussi définir une fonction à l'intérieur d'une fonction. Par exemple, on peut définir f : $x \longmapsto 2x + \sqrt{2(x+1)}$ en utilisant une fonction locale $g : y \longmapsto 2y$:

[32]:
```
let f x =
    let g y = 2.*.y in (* g n'est utilisable que dans f *)
    g x +. (g (x +. 1.))**0.5
```

[32]:
```
val f : float -> float = <fun>
```

[33]:
```
f 1.
```

[33]:
```
- : float = 4.
```

Exercice : Écrire une fonction `swap` qui échange les valeurs de 2 références en argument.
`swap` doit être de type `'a ref -> 'a ref -> unit`, ce qui signifie que `swap` a deux références en argument, sur des valeurs de même type `'a`, et ne renvoie pas de valeur.
On rappelle les opérations sur les références :
- Définir une référence (locale) : `let a = ref 5 in ...` - Obtenir la valeur d'une référence : `!a`
- Modifier une référence : `a := 7`

Remarque importante : Lorsque l'on modifie une référence (ou un autre objet impératif, comme un tableau) qui est l'argument d'une fonction, on la modifie aussi à l'extérieur de la fonction. C'est ce qu'on appelle un **passage par référence**.

[34]:
```
(* 2. *)
let swap x y =
    let tmp = !y in
    y := !x;
    x := tmp;;

let x = ref 1 in
let y = ref 2 in
swap x y;
x, y;;
```

[34]:
```
val swap : 'a ref -> 'a ref -> unit = <fun>
```

[34]:
```
- : int ref * int ref = ({contents = 2}, {contents = 1})
```

2.8 Booléens

Une valeur booléenne (bool) est soit true (vrai) soit false (faux). Exemple de variable de type bool :

[1]:
```
let a = true
```

[1]:
```
val a : bool = true
```

2.9 Comparaison

On peut tester l'égalité de deux **valeurs** avec = :

[2]:
```
3 = 1 + 2
```

[2]:
```
- : bool = true
```

On peut aussi utiliser = sur des variables, auquel cas on compare leurs valeurs :

[3]:
```
let a = 3 in
let b = 4 in
a = b (* teste si les valeurs de a et b sont égales *)
```

[3]:
```
- : bool = false
```

Attention : le = de OCaml correspond au == de Python. == existe en OCaml, mais compare les **adresses mémoires** de 2 variables au lieu de leurs valeurs et on ne l'utilisera presque jamais.

On peut comparer des valeurs numériques avec < (inférieur strict), >, <= (inférieur ou égal), >=, <> (différent)… :

[4]:
```
2 < 1
```

[4]:
```
- : bool = false
```

Il faut obligatoirement comparer des valeurs de même type :

[5]:
```
2.4 < 3 (* on ne peut pas comparer un float avec un int *)
```

```
File "[5]", line 1, characters 6-7:
1 | 2.4 < 3 (* on ne peut pas comparer un float avec un int *)
          ^

Error: This expression has type int but an expression was expected of type
         float
```

[6]:
```
2.4 < 3.0 (* par contre ceci fonctionne *)
```

[6]:
```
- : bool = true
```

Remarque : pas besoin de mettre des points (.) sur `<`, `>` …

Les opérateurs `&&` (et), `||` (ou), `not` permettent de combiner des conditions :

[7]:
```
1 < 2 && 2 < 3
```

[7]:
```
- : bool = true
```

[8]:
```
let a = 0 in
a <> 0 || a > 3 (* test si a est différent de 0 ou supérieur à 3 *)
```

[8]:
```
- : bool = false
```

Exercice

1. Quelle est la valeur du code suivant?

```
let a = 42 in
not (a = 42 && (a < 10 || a > 30))
```

2. Comment aurait-on pu écrire `not (a = 42 && (a < 10 || a > 30))` sans `not`?
3. Écrire une fonction `xor : bool -> bool -> bool` telle que `xor a b` renvoie le "ou exclusif" de `a` et `b`, c'est à dire `true` si `a` ou `b` est `true`, mais pas les deux.

2.10 Condition if

On peut écrire une condition `if` de la façon suivante en OCaml :

```
if ... then ... else ...
```

La condition du `if` doit être un booléen. Si la condition est vraie, le `then` est exécuté et sa valeur est renvoyé. Sinon, c'est la valeur du `else` qui est renvoyé :

[9]:
```
if 1 = 2 then 42 else 24
```

[9]:
```
- : int = 24
```

Définissons par exemple la fonction valeur absolue ($x \longmapsto |x|$) :

[10]:
```
let abs x =
    if x < 0. then -. x
    else x
```

[10]:
```
val abs : float -> float = <fun>
```

Rappelons qu'il n'y a pas de `return` en OCaml : c'est la dernière expression calculée par la fonction qui est renvoyée. Ainsi `abs x` renvoie `-x` si `x` est négatif et `x` sinon.

[11]:
```
abs (-2.718)  (* je mets des parenthèses à cause du - *)
```

[11]:
```
- : float = 2.718
```

Pour plus de lisibilité on sautera une ligne avant `then` et `else`, sauf si le contenu du `if` est très court.

Dans un `if ... then ... else ...`, les valeurs dans le `then` et dans le `else` doivent être de même type :

[12]:
```
if 1 = 1 then 2
else 3.14  (* impossible d'avoir 2 types différents dans le then et else *)
```

```
File "[12]", line 2, characters 5-9:
2 | else 3.14  (* impossible d'avoir 2 types différents dans le then et else *)
         ^^^^
Error: This expression has type float but an expression was expected of type
         int
```

La valeur renvoyée par `if ... then ... else ...` peut être stockée dans une variable :

[13]:
```
let a = -5 in
let b = if a > 0 then a else -a in (* on pourrait aussi calculer une valeur␣
 ↪absolue comme ça *)
b  (* b vaut 5 *)
```

[13]:
```
- : int = 5
```

Exercice Définir les fonctions suivantes en OCaml :
1.

2.

3.

Exercice Écrire une fonction `n_solutions : float -> float -> float -> int` telle que `n_solutions a b c` renvoie le nombre de solutions de l'équation $ax^2 + bx + c$.

3 Récursivité

La récursivité est la possibilité pour une fonction de s'appeller soi-même. En général, il y a deux étapes pour écrire une fonction récursive : 1. Un **cas de base** où la fonction renvoie directement une valeur. 2. Un **cas général** où la fonction s'appelle sur des paramètres "plus petits".

En OCaml, une fonction récursive doit être définie par `let rec ...`. Voici un exemple :

[1]:
```
let rec f x = (* exemple de fonction récursive *)
    if x = 0 then print_newline () (* cas de base *)
    else (print_int x;
          f (x - 1)) (* cas général *)
```

[1]:
```
val f : int -> unit = <fun>
```

`f x` affiche un retour à la ligne si `x` est égal à 0, et sinon affiche `x` puis appelle `f (x - 1)`.

Essayons cette fonction :

[2]:
```
f 2
```

```
21
```

[2]:
```
- : unit = ()
```

Voici ce qui se passe lors de cet appel `f 2` :

1. On regarde si `2 = 0`, ce qui est faux. On passe donc dans le `else`.
2. On affiche `2` avec `print_int x`.
3. On appelle `f` sur la valeur 1. Le calcul de `f 2` se met en pause et on exécute `f 1`. Quand `f 1` sera terminé, l'appel de `f 2` continuera et `f 1` sera remplacé par sa valeur de retour.
4. L'exécution de `f 1` affiche `1` puis appelle `f 0`. Le calcul de `f 1` se met en pause et on exécute `f 0`. Quand `f 0` sera terminé, l'appel de `f 2` continuera et `f 0` sera remplacé par sa valeur de retour.
5. `f 0` exécute `print_newline ()` et s'arrête (en renvoyant `()`).
6. L'exécution de `f 1` reprend et `f 1` s'arrête.
7. L'exécution de `f 2` reprend et `f 2` s'arrête.

Vous pouvez visualiser l'exécution d'un code similaire en Python avec Python Tutor. Il est important de bien comprendre comment les appels récursifs s'effectuent.

Un exemple classique d'utilisation de la récursivité est le calcul de la factorielle d'un entier n, définie par $n! = n \times (n-1) \times ... \times 2 \times 1$.
Pour définir une fonction récursive calculant $n!$ on a besoin de deux choses : - **Cas de base** : si $n = 0$, on peut renvoyer directement 1 (par convention $0! = 1$), sans appel récursif. - **Cas général/récurrence** : si n est quelconque, il faut rammener le calcul de $n!$ à un calcul d'une factorielle plus petite (de façon à se rapprocher du cas de base). Pour cela, on peut remarquer que $n! = n \times (n-1)!$ et donc que calculer $(n-1)!$ permet d'en déduire $n!$.
On en déduit le code suivant :

[3]:
```
let rec fact n =
    if n = 0 then 1 (* par convention 0! = 1 *)
    else n*fact (n - 1)
```

[3]:
```
val fact : int -> int = <fun>
```

[4]:
```
fact 4
```

[4]:
```
- : int = 24
```

Remarque : Si on oublie le cas de base (`if n = 0`) la fonction ne s'arrête jamais (`fact 0` appelerait `fact (-1)` qui appelerait `fact (-2)` et ainsi de suite…) !

Écrire une fonction récursive ressemble beaucoup à écrire une démonstration mathématiques par récurrence et d'ailleurs on utilisera souvent une démonstration par récurrence pour démontrer qu'une fonction récursive est correcte, c'est à dire renvoie bien la bonne valeur.
Par exemple, pour démontrer que `fact` est correcte, on peut poser l'hypothèse de récurrence :

$$\mathcal{H}(n) : \text{fact } n \text{ renvoie } n!$$

Preuve : 1. $\mathcal{H}(0)$ est vraie car `fact 0` renvoie 1 et, par définition, $0! = 1$. 1. Soit n un entier strictement positif. Supposons $\mathcal{H}(n-1)$ et montrons $\mathcal{H}(n)$.
Comme $n > 0$, `fact n` renvoie `n*fact (n - 1)`. D'après $\mathcal{H}(n-1)$, `fact (n - 1)` renvoie $(n-1)!$.
Donc `fact n` renvoie $n(n-1)! = n!$, ce qui démontre $\mathcal{H}(n)$.
D'après le principe de récurrence, $\mathcal{H}(n)$ est vraie pour tout $n \in \mathbb{N}$.

Exercice : Qu'affiche le code suivant? Le deviner puis exécuter le code pour vérifier.

```
let rec f x =
    if x = 0 then print_newline ()
    else (f (x - 1);
          print_int x) in
f 5
```

Exercice : 1. Écrire une fonction récursive pour calculer la somme des n premiers entiers $S(n) = 0 + 1 + 2 + ... + (n-1)$. 2. Quelle formule connaissez-vous pour calculer $S(n)$? En déduire une autre fonction (non récursive) pour calculer cette valeur. Vérifier sur des exemples que les deux fonctions donnent la même valeur.

Une application classique de la récursivité est le calcul des termes d'une suite récurrente.
Par exemple :

$$\begin{cases} u_n = 3u_{n-1} + 2, \text{si } n > 0 \\ u_0 = 5 \end{cases}$$

Cette définition par récurrence se traduit naturellement en fonction récursive :

[5]:
```
let rec u n =
    if n = 0 then 5
    else 3*(u (n - 1)) + 2
```

[5]:
```
val u : int -> int = <fun>
```

[6]:
```
u 10
```

[6]:
```
- : int = 354293
```

Exercice : calculer v_{10}, où v_n est définie par

$$\begin{cases} v_{n+1} = \sqrt{v_n} + 4, \text{si } n > 0 \\ v_1 = 5 \end{cases}$$

Un autre exemple classique est la suite de Fibonacci :

$$u_0 = 1$$

$$u_1 = 1$$

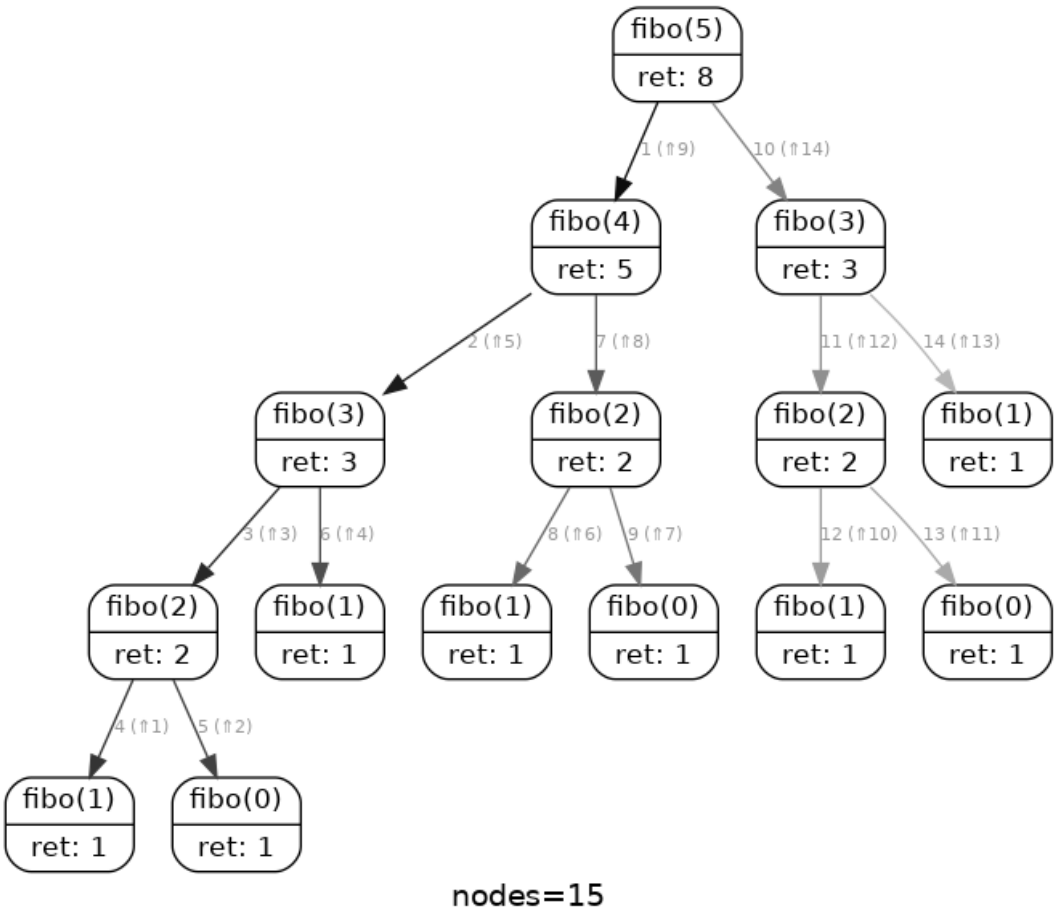
$$u_n = u_{n-1} + u_{n-2}$$

On pourrait l'implémenter de la façon suivante :

[7]:
```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

[7]:
```
- : int = 89
```

Attention : cette méthode est très inefficace. Pour s'en convaincre, regardons visuellement les appels récursifs de fibo :



Il y a de nombreux calculs inutiles : par exemple, `fibo 3` est appelé 2 fois et `fibo 2` est appelé 3 fois, ce qui est inefficace.

Exercice : Montrer que le nombre d'appels récursifs pour calculer `fibo n` est exponentiel en n (c'est à dire supérieur à a^n pour un certain a indépendant de n).

Soit C_n le nombre d'appels récursifs effectués par `fibo n` (on compte l'appel de `fibo n` ainsi que tous ses sous-appels récursifs). Alors :

$$C_n = \underbrace{C_{n-1}}_{\text{appels récursifs de fibo (n-1)}} + \underbrace{C_{n-2}}_{\text{appels récursifs de fibo (n-2)}}$$

C_n vérifie donc la même équation de récurrence que la suite de Fibonacci. Pour donner un minorant simplement, on peut remarquer que $C_{n-1} \geq C_{n-2}$ (car $C_{n-1} = C_{n-2} + \underbrace{C_{n-3}}_{\geq 0}$) donc $C_n \geq 2C_{n-2}$ et en appliquant cette inégalité plusieurs fois :

$$C_n \geq 2C_{n-2} \geq 2^2 C_{n-4} \geq 2^3 C_{n-6} \geq ... \geq 2^{\frac{n}{2}} C_{n-2\frac{n}{2}} = 2^{\frac{n}{2}}$$

On a donc montré que $C_n \geq 2^{\frac{n}{2}}$: `fibo n` effectue un nombre exponentiel (en n) d'appels récursifs.

Il est possible d'éviter ces appels inutiles en utilisant un **accumulateur**. Un accumulateur est un argument d'une fonction récursive que l'on va utiliser pour construire le résultat final. L'accumulateur est modifié à chaque appel récursif.

[8]:
```
let rec fibo2 n a b =
  if n = 0 then b
  else fibo2 (n - 1) (a + b) a in
fibo2 10 1 1
```

[8]:
```
- : int = 89
```

On verra aussi plus tard une technique de **mémoïsation** permettant d'éviter de faire 2x le même appel récursif, de façon systématique.

Bien sûr, on pourrait aussi utiliser une boucle `for` en stockant les deux derniers termes de la suite dans des variables, mais l'objectif ici est de s'entraîner à penser récursivement.

3.1 Sous-fonction récursive

Quand on souhaite écrire une fonction `f x` en utilisant une méthode récursive mais que `x` doit être accessible dans les appels récursifs, on peut utiliser une sous-fonction récursive dans `f`, et `f` se contentera d'appeler cette fonction.

Par exemple, pour savoir si un entier est premier :

[9]:
```
let premier n =
    let rec f k =  (* renvoie true si n n'a pas de diviseurs entre 2 et k *)
        if k = 1 then true  (* on a regardé tous les diviseurs potentiels *)
        else if n mod k = 0 then false  (* si k divise n *)
        else f (k - 1) in (* vérifie que n n'a pas de diviseurs entre 2 et k *)
    f (n - 1)  (* teste si n a un diviseur entre 2 et n - 1 *)
```

[9]:
```
val premier : int -> bool = <fun>
```

[10]:
```
premier 2 && premier 3 && not (premier 4)  (* test *)
```

[10]:
```
- : bool = true
```

[ ]:
```

```

Comme en Python, OCaml a deux boucles permettants de répéter des instructions : `for` et `while`.

4 Boucle for

Pour répéter `instructions` pour des valeurs de `i` allant de `a` à `b` **inclus** (contrairement à Python) :

```
for i=a to b do
    instructions
done
```

[1]:
```
for i=0 to 5 do
    print_int i
done;
print_newline()
```

```
012345
```

[1]:
```
- : unit = ()
```

Exercice : Combien de fois est répété `for i=a to b do ...`? (c'est-à-dire : combien y a t-il d'entiers de a à b inclus?)

Exercice : Écrire une fonction pour calculer la somme des carrés des n premiers entiers en utilisant un `for`, puis une fonction récursive.

Comme on le voit sur l'exercice précédent, il est général plus clair et concis d'écrire une fonction récursive en OCaml. De manière générale, il ne faut pas abuser des références et boucles et s'entraîner à penser et écrire récursivement.

Une variante de la boucle `for` avec `downto` permet d'énumérer "à l'envers" :

[2]:
```
for i=5 downto 0 do
    print_int i
done;
print_newline()
```

```
543210
```

[2]:
```
- : unit = ()
```

5 Boucle while

Pour répéter `instructions` tant que `condition` est vraie :

```
while condition do
    instructions
done
```

En guise d'illustration, considérons l'algorithme d'Euclide pour le calcul du PGCD de deux entiers a et b. Cet algorithme consiste à répéter les opérations suivantes tant que $b \neq 0$: - Calculer le reste r de la division euclidienne de a par b. - Remplacer a par b et b par r.

Quand $b = 0$, on peut montrer que la valeur de a est le PGCD de a et b.

Voici le code OCaml correspondant avec une boucle `while` :

[3]:
```
let pgcd a b =
    let q = ref a in
    let r = ref b in
```

```
    while !r <> 0 do
        let tmp = !q in (* on a besoin de l'ancienne valeur de q pour calculer
↪le nouveau r *)
        q := !r;
        r := tmp mod !r
    done;
    !q;;

pgcd 30 12;;
```

[3]:
```
val pgcd : int -> int -> int = <fun>
```

[3]:
```
- : int = 6
```

Voici ce que cela donnerait avec une fonction récursive (encore une fois c'est beaucoup plus simple en récursif!) :

[4]:
```
let rec pgcd a b =
    if b = 0 then a
    else pgcd b (a mod b);;

pgcd 30 12;;
```

[4]:
```
val pgcd : int -> int -> int = <fun>
```

[4]:
```
- : int = 6
```

Exercice
Soit $a \in \mathbb{N}$. La suite de Syracuse est définie par $s_0 = a$ et

$$s_{n+1} = \begin{cases} \frac{s_n}{2}, \text{si } n \text{ est pair} \\ 3s_n + 1, \text{sinon} \end{cases}$$

Écrire une fonction `temps_vol` ayant a en argument et renvoyant le premier indice n tel que $s_0 = a$ et $s_n = 0$.

[5]:
```
let temps_vol a =
    let t = ref 0 in
    let s = ref a in
    while !s <> 1 do
        incr t;
        s := if !s mod 2 = 0 then !s/2 else 3* !s + 1
    done;
    !t in
```

```
temps_vol 10
```

[5]:
```
- : int = 6
```

[ ]:
```

```