OCaml

I Variables

I.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 `;;`.

I.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 (`...`)

[4]:
```
let y = -1 in y (* y est accessible dans le in *)
```

[4]:
```
- : int = -1
```

[5]:
```
y (* utiliser y ici donne une erreur  *)
```

```
File "[5]", 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 $54a$. **Remarque** Il est possible d'utiliser des parenthèses ou `begin ... end` pour délimiter le in :

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

I.3 Opérations numériques

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

[6]:
```
3 * 14
```

[6]:
```
- : int = 42
```

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

[7]:
```
3 / 2 (* division entière *)
```

[7]:
```
- : 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) :

[8]:
```
let pi = 3.141592
```

[8]:
```
val pi : float = 3.141592
```

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

[9]:
```
pi +. 2.618 (* noter le . après + *)
```

[9]:
```
- : 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 `**` :

[10]:
```
2.718 ** 3.14 (* x puissance y *)
```

[10]:
```
- : 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 `**` :

[11]:
```
2. ** 10.
```

[11]:
```
- : 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. **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.

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?

I.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 :

[12]:
```
max_int
```

[12]:
```
- : int = 4611686018427387903
```

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

[13]:
```
max_int + 1
```

[13]:
```
- : 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 :

[14]:
```
max_float
```

[14]:
```
- : 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).

[15]:
```
2.**0.5 (* seulement une partie des décimales est stockée *)
```

[15]:
```
- : float = 1.41421356237309515
```

[16]:
```
(2.**0.5)**2.0  (* ne donne pas 2.0 à cause des erreurs d'arrondis *)
```

[16]:
```
- : 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 :

[17]:
```
0.1 +. 0.2  (* ne donne pas 0.3 *)
```

[17]:
```
- : float = 0.30000000000000044
```

`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 :

[18]:
```
2.**1023.
```

[18]:
```
- : float = 8.98846567431158e+307
```

[19]:
```
2.**1024.
```

[19]:
```
- : float = infinity
```

I.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 :

[20]:
```
print_int 42
```

[20]:
```
- : unit = ()
```

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

[21]:
```
print_newline ()
```

```
42
```

[21]:
```
- : 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 :

[22]:
```
print_int 31;
print_newline ()
```

```
31
```

[22]:
```
- : unit = ()
```

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

[23]:
```
let a = 3 in
print_int a;
a (* a reste accessible : il est toujours dans le in *)
```

[23]:
```
- : int = 3
```

[24]:
```
let a = 3 in
print_int a;;
a;; (* a n'est pas accessible ici *)
```

[24]:
```
- : unit = ()
```

[24]:
```
- : int = 10
```

I.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 :

[25]:
```
let a
  =
      6 (* indentation n'importe comment *)
```

[25]:
```
val a : int = 6
```

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

I.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** :

[26]:
```
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 *)
```

[26]:
```
val x : int = 3
```

[26]:
```
- : bool = false
```

[26]:
```
- : 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 *)
```

[27]:
```
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 *)
```

[27]:
```
val x : int ref = {contents = 3}
```

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

[27]:
```
- : int = 4
```

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
```

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

II Fonctions

II.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 :

[28]:
```
sqrt 2.0 (* renvoie une approximation de racine de 2 *)
```

[28]:
```
- : 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.

[29]:
```
sqrt
```

[29]:
```
- : 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 :

[30]:
```
sqrt 2 (* erreur : on donne un entier à sqrt qui attend un flottant *)
```

```
File "[30]", 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
```

II.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 $f : x \longmapsto 2x$:

[31]:
```
let f x = 2*x
```

[31]:
```
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 :

[32]:
```
f 3
```

[32]:
```
- : 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 :

[33]:
```
let f y = 2*y (* peu importe le nom de la variable muette y *)
```

[33]:
```
val f : int -> int = <fun>
```

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

[34]:
```
f 3
```

[34]:
```
- : int = 6
```

Exercice : définir la fonction $f : x \longmapsto \frac{1}{\sqrt{1+x^2}}$ en OCaml. Comme pour les variables, il est possible d'utiliser `in` pour spécifier la portée d'une fonction g

[35]:
```
let g x = x + 1 in
g 0  (* g est utilisable seulement dans le in *)
```

[35]:
```
- : int = 1
```

Exercice Donner la valeur de l'expression suivante :

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

II.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.

[36]:
```
fun x -> x*2 (* définition d'une fonction anonyme *)
```

[36]:
```
- : int -> int = <fun>
```

[37]:
```
(fun x -> x*2) 3 (* applique une fonction anonyme sur la valeur 3 *)
```

[37]:
```
- : 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 :

[38]:
```
let f = fun x -> 2.0*.x**0.5
```

[38]:
```
val f : float -> float = <fun>
```

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

II.4 Fonctions de plusieurs variables

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

[39]:
```
let sum x y = x + y
```

[39]:
```
val sum : int -> int -> int = <fun>
```

[40]:
```
sum 3 4 (* renvoie 3 + 4 *)
```

[40]:
```
- : 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 :

[41]:
```
sum 3
```

[41]:
```
- : 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 :

[42]:
```
let f = sum 3 in (* f est une fonction *)
f 4 (* renvoie sum 3 4, c'est à dire 7 *)
```

[42]:
```
- : 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**) :

[43]:
```
let sum = fun x -> (fun y -> x + y) (* OCaml transforme la définition de sum␣
↪ci-dessus en celle-ci *)
```

[43]:
```
val sum : int -> int -> int = <fun>
```

[44]:
```
(sum 2) 3  (* le calcul effectué par OCaml lorsqu'on écrit sum 2 3 *)
```

[44]:
```
- : 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 `... -> ... -> ... -> ....` 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 :

[45]:
```
print_int 0;
print_newline ();
print_int 1;
print_newline ();
```

```
330
1
```

[45]:
```
- : unit = ()
```

II.5 Polymorphisme

Reprenons notre 1er exemple de fonction :

[46]:
```
let f x = 2*x
```

[46]:
```
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 :

[47]:
```
let id x = x
```

[47]:
```
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
```

[48]:
```
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 *)
```

[48]:
```
val f : 'a -> 'b -> 'b = <fun>
```

[49]:
```
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 -> ()*)
```

[49]:
```
val g : int -> 'a -> ('a -> int) -> int = <fun>
```

II.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 :

[50]:
```
let eval f =
f 0
```

[50]:
```
val eval : (int -> 'a) -> 'a = <fun>
```

[51]:
```
let f x = 3*x + 7 in
eval f
```

[51]:
```
- : 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?

II.7 Variable locale à une fonction

Il est possible de définir une variable dans une fonction :

[52]:
```
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 *)
```

[52]:
```
val pow4 : int -> int = <fun>
```

[53]:
```
pow4 2 (* test de notre fonction *)
```

[53]:
```
- : 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 $g : y \longmapsto 2y$:

[54]:
```
let f x =
    let g y = 2.*.y in (* g n'est utilisable que dans f *)
    g x +. (g (x +. 1.))**0.5
```

[54]:
```
val f : float -> float = <fun>
```

[55]:
```
f 1.
```

[55]:
```
- : 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**.

II.8 Booléens

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

[56]:
```
let a = true
```

[56]:
```
val a : bool = true
```

II.9 Comparaison

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

[57]:
```
3 = 1 + 2
```

[57]:
```
- : bool = true
```

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

[58]:
```
let a = 3 in
let b = 4 in
a = b (* teste si les valeurs de a et b sont égales *)
```

[58]:
```
- : 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)… :

[59]:
```
2 < 1
```

[59]:
```
- : bool = false
```

Il faut obligatoirement comparer des valeurs de même type :

[60]:
```
2.4 < 3 (* on ne peut pas comparer un float avec un int *)
```

```
File "[60]", 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
```

[61]:
```
2.4 < 3.0 (* par contre ceci fonctionne *)
```

[61]:
```
- : bool = true
```

Remarque : pas besoin de mettre des points (.) sur <, > … Les opérateurs && (et), || (ou), not permettent de combiner des conditions :

[62]:
```
1 < 2 && 2 < 3
```

[62]:
```
- : bool = true
```

[63]:
```
let a = 0 in
a <> 0 || a > 3 (* test si a est différent de 0 ou supérieur à 3 *)
```

[63]:
```
- : 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.

II.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 :

[64]:
```
if 1 = 2 then 42 else 24
```

[64]:
```
- : int = 24
```

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

[65]:
```
let abs x =
    if x < 0. then -. x
    else x
```

[65]:
```
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.

[66]:
```
abs (-2.718)  (* je mets des parenthèses à cause du - *)
```

[66]:
```
- : float = 2.718
```

Pour plus de lisibilité on sautera une ligne avant `then` et `else`, sauf si le contenu du `if` est très court.
Exercice : Écrire une fonction `max` renvoyant le maximum de ses 2 arguments.
Remarque : cette fonction existe déjà et peut être utilisée telle quelle.

[67]:
```
let max x y = if x < y then y else x
```

[67]:
```
val max : 'a -> 'a -> 'a = <fun>
```

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

[68]:
```
if 1 = 1 then 2
else 3.14  (* impossible d'avoir 2 types différents dans le then et else *)
```

```
File "[68]", 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 :

[69]:
```
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 *)
```

[69]:
```
- : 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 $ax^2 + bx + c$.

III 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 :

[70]:
```
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 *)
```

[70]:
```
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 :

[71]:
```
f 2
```

```
21
```

[71]:
```
- : 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 :

[72]:
```
let rec fact n =
    if n = 0 then 1 (* par convention 0! = 1 *)
    else n*fact (n - 1)
```

[72]:
```
val fact : int -> int = <fun>
```

[73]:
```
fact 4
```

[73]:
```
- : 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, $0! = 1$.
2. 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 :

[74]:
```
let rec u n =
    if n = 0 then 5
    else 3*(u (n - 1)) + 2
```

[74]:
```
val u : int -> int = <fun>
```

[75]:
```
u 10
```

[75]:
```
- : 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 :

$$u_0 = 1$$

$$u_1 = 1$$

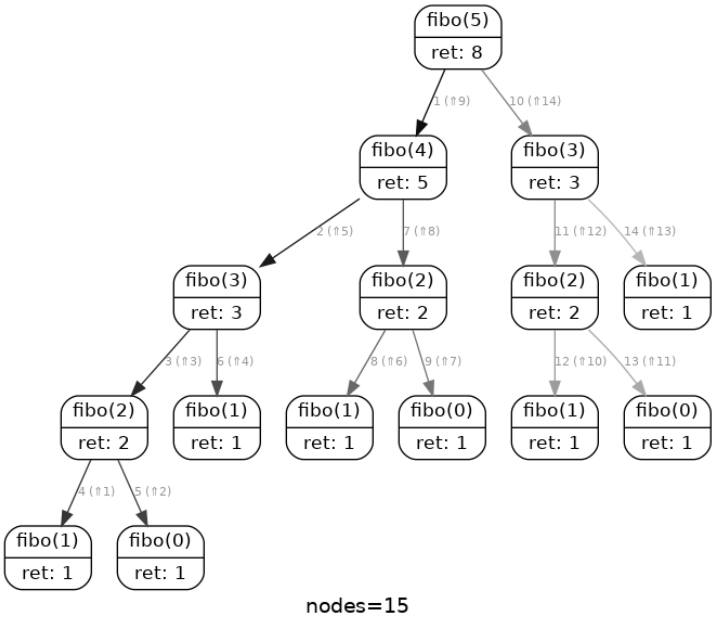
$$u_n = u_{n-1} + u_{n-2}$$

On pourrait l'implémenter de la façon suivante :

[76]:
```
let rec fibo n =
    if n <= 1 then 1
    else fibo (n - 1) + fibo (n - 2) in
fibo 10
```

[76]:
```
- : int = 89
```

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



nodes=15

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 :

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

On a donc montré que $\boxed{\forall n \in \mathbb{N}, C_n \geq 2^{\frac{n}{2}}}$: `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.

[77]:
```
let rec fibo2 n a b =
  (* n est le nombre de termes restants à calculer *)
  (* a est le dernier terme calculé de la suite *)
  (* b est l'avant-dernier terme calculé *)
  if n = 0 then b
  else fibo2 (n - 1) (a + b) a in (* les derniers termes deviennent a+b et a *)
fibo2 10 1 1
```

[77]:
```
- : 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.

III.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 :

[78]:
```
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 *)
```

[78]:
```
val premier : int -> bool = <fun>
```

[79]:
```
premier 2 && premier 3 && not (premier 4)  (* test *)
```

[79]:
```
- : bool = true
```

IV Structures de données persistantes

Une **structure de donnée** permet de stocker un ensemble d'éléments et de faire certaines opérations sur ces éléments.
Une structure est persistante (ou encore : non mutable) s'il est impossible de modifier ses éléments. OCaml possède principalement 3 structures persistantes : - **tuple**, ou n-uplets - **list**, les plus utilisées - **string**, pour les chaînes de caractères

IV.1 n-uplet (tuple)

Un n-uplet est défini comme en mathématiques. Par exemple, un 2-uplet est un couple (qui réprésente par exemple un point dans le plan $\mathbb{R}^2$:

[80]:
```
let point = (1.75, 2.5)  (* un couple de 2 flottants *)
```

[80]:
```
val point : float * float = (1.75, 2.5)
```

OCaml nous répond que `point` est de type `float*float`, ce qui signifie un couple de 2 `float`.
On peut récupérer les coordonnées d'un n-uplet de la façon suivante :

[81]:
```
let (a, b) = point  (* met la 1ère coordonnée de point dans a et la 2ème dans b *)
```

[81]:
```
val a : float = 1.75
val b : float = 2.5
```

Dans le cas d'un couple, on peut aussi récupérer le 1er et 2ème élement avec les fonctions `fst` et `snd` :

[82]:
```
fst point;;
snd point;;
```

[82]:
```
- : float = 1.75
```

[82]:
```
- : float = 2.5
```

Exercice À votre avis, quels sont les types de `fst` et `snd` ? Vérifier avec OCaml. Dans une fonction, on peut aussi décomposer directement le tuple en argument de la fonction :

[83]:
```
let mid (x1, y1) (x2, y2) = (* renvoie le milieu des deux points *)
  (x1 +. x2) /. 2., (y1 +. y2) /. 2.
```

[83]:
```
val mid : float * float -> float * float -> float * float = <fun>
```

Les parenthèses sont en fait faculatitives autour d'un tuple : l'instruction précédente est donc équivalente à `let a, b = point`. De plus, attention à utiliser `.` et pas `,` pour des flottants, sinon on obtient un tuple :

[84]:
```
3,14  (* Attention : c'est un tuple et non pas un flottant *)
```

[84]:
```
- : int * int = (3, 14)
```

Un n-uplets peut contenir des contenir des éléments de types différents :

[85]:
```
let t = (1, 2.2, true)  (* un triple contenant un entier, un flottant et un booléen␣
↪*)
```

[85]:
```
val t : int * float * bool = (1, 2.2, true)
```

Deux tuples sont égaux s'ils sont de même taille et que les composantes sont égales 2 à 2 :

[86]:
```
(1, 2) = (1, 2);;
(1, 2) = (1, 3);;
```

[86]:
```
- : bool = true
```

[86]:
```
- : bool = false
```

Exercice : On représente un nombre complexe par un couple de flottants (composé de la partie réelle et partie imaginaire). 1. Définir $1 - 2i$ sous forme de couple.
2. Définir une fonction `conjugue : float*float -> float*float` renvoyant le conjugué $\bar{z}$ d'un nombre complexe z.
3. Écrire une fonction `add` qui prend deux nombres complexes z_1 et z_2 en arguments et renvoie $z_1 + z_2$.
4. Écrire une fonction `mul` qui multiplie deux nombres complexes en arguments.
5. Écrire une fonction `div` qui divise deux nombres complexes en arguments (on utilisera la multiplication par le conjugué : $\frac{a+ib}{c+id} = \frac{(a+ib)(c-id)}{(c+id)(c-id)} = \frac{ac+bd+i(bc-ad)}{c^2-d^2}$ Une des utilités des tuples et de permettre de renvoyer plusieurs résultats par une fonction (on les renvoie sous forme de tuple).

IV.2 Listes

Une liste se définie avec des crochets (`[...]`), les éléments étant séparés par des point-virgules (`;`) :

[87]:
```
[1; 7; -1]  (* liste composée de 3 entiers *)
```

[87]:
```
- : int list = [1; 7; -1]
```

OCaml nous indique qu'il s'agit d'une valeur de type `int list`, c'est à dire liste d'entiers. Voici d'autres exemples de listes :

[88]:
```
[3.14; 2.718]  (* liste de 2 flottants *)
```

[88]:
```
- : float list = [3.14; 2.718]
```

[89]:
```
[]  (* liste vide *)
```

[89]:
```
- : 'a list = []
```

Par contre, on ne peut pas avoir plusieurs types différents dans la même liste :

[90]:
```
[3.14; 2]
```

```
File "[90]", line 1, characters 7-8:
1 | [3.14; 2]
           ^

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

L'instruction `e::l` renvoie une liste obtenue à partir de l en ajoutant l'élément e au début :

[91]:
```
1::[2; 3]  (* ajoute 1 au début de la liste [2; 3] *)
```

[91]:
```
- : int list = [1; 2; 3]
```

Attention : cons (`::`) renvoie une nouvelle liste, mais ne modifie pas celle à droite. Par exemple :

[92]:
```
let l = [1; 2; 3];;
0::l;;  (* donne une nouvelle liste *)
l;;  (* l n'a pas été modifiée *)
```

[92]:
```
val l : int list = [1; 2; 3]
```

[92]:
```
- : int list = [0; 1; 2; 3]
```

[92]:
```
- : int list = [1; 2; 3]
```

Si on veut ajouter un élément à une liste, il faut donc construire une nouvelle liste :

[93]:
```
let l2 = 0::l;;  (* l2 est une nouvelle liste obtenue à partir de l en rajoutant 0 *)
```

[93]:
```
val l2 : int list = [0; 1; 2; 3]
```

Attention On ne peut pas ajouter simplement d'élément à la fin d'une liste (il faudrait parcourir récursivement chaque élément de la liste jusqu'à la fin, ce qui est inefficace). On peut se servir de `::` pour construire une liste élément par élément, avec une fonction récursive :

[94]:
```
let rec range n =  (* renvoie la liste des entiers de 1 à n (à l'envers) *)
  if n = 0 then []  (* cas de base *)
  else n::range (n-1);;
range 5
```

[94]:
```
val range : int -> int list = <fun>
```

[94]:
```
- : int list = [5; 4; 3; 2; 1]
```

Exercice : Écrire une fonction `pairs : int -> int list` telle que `pairs n` renvoie la liste des entiers pairs entre 0 et $2n$ (inclus).

IV.3 Pattern matching (filtrage)

IV.3.1 Pattern matching simple

La façon classique de parcourir une liste `l` en OCaml est d'utiliser un **pattern matching**, qui consiste à regarder la forme de `l` : - soit `l` est vide (cas de base) - soit `l` contient un premier élément (la tête), puis le reste de la liste (la queue) Voici la syntaxe OCaml :

[95]:
```
let l = [1; 2; 3]
```

[95]:
```
val l : int list = [1; 2; 3]
```

[96]:
```
match l with
  | [] -> 0  (* si l est vide, on renvoie 0 *)
  | e::q -> e  (* sinon l est de la forme e::q, on renvoie e *)
```

[96]:
```
- : int = 1
```

Comme `l` est non-vide, on passe dans le 2ème cas du match et on affiche le premier élément de `l`. Essayez de mettre une liste vide à la place de `l` dans le `match` précédent pour voir la différence. La plupart du temps, on utilise un `match` dans une fonction récursive ayant une liste en argument. Voici par exemple une fonction récursive pour calculer le nombre d'éléments d'une liste :

[97]:
```
let rec taille l = match l with
  | [] -> 0  (* une liste vide est de taille 0 *)
  | e::q -> 1 + taille q;;  (* sinon l contient e + tous les éléments de q *)
taille l;;  (* vérification *)
```

[97]:
```
val taille : 'a list -> int = <fun>
```

[97]:
```
- : int = 3
```

Remarque : OCaml nous dit que `taille` est de type `'a list -> int`. `'a` signifie "n'importe quel type". Il n'y a donc pas de contrainte sur le type des éléments de la liste `l` en argument. **Remarque** : quand une fonction possède un seul argument qui est une liste, on peut écrire `let f = function ...` au lieu de `let f l = match l with ...`. Par exemple on peut réécrire la fonction `taille` précédente :

[98]:
```
let rec taille = function  (* même chose que let taille l = match l with ... *)
  | [] -> 0
  | e::q -> 1 + taille q;;
```

[98]:
```
val taille : 'a list -> int = <fun>
```

Exercice : 1. Écrire une fonction `somme : int list -> int` pour calculer la somme des termes d'une liste. Par exemple, `somme [4; 7; 3]` doit renvoyer 14. 2. Réutiliser la fonction `range` ci-dessus pour calculer la somme des entiers de 1 à n. **Exercice** : Écrire une fonction `maximum` permettant de renvoyer le plus grand élément d'une liste d'entiers.

IV.3.2 Pattern matching plus avancé

Un pattern matching peut aussi permettre de décomposer sous d'autres formes :

[99]:
```
let deuxieme l = match l with  (* renvoie le 2ème élément d'une liste  *)
  | e1::e2::q -> e2;;  (* décompose la liste en le 1er élément e1, le 2ème e2 et la␣
 ↪liste des autres éléments *)
deuxieme [4; 7; 3]
```

```
File "[99]", line 1, characters 17-96:
1 | …match l with  (* renvoie le 2ème élément d'une liste  *)
2 |   | e1::e2::q -> e2…
…
Warning 8: this pattern-matching is not exhaustive.
Here is an example of a case that is not matched:
(_::[]|[])
```

[99]:
```
val deuxieme : 'a list -> 'a = <fun>
```

[99]:
```
- : int = 7
```

Remarque : OCaml nous dit que notre pattern matching ne couvre pas tous les cas. En effet, notre fonction ne dit pas quoi faire dans le cas où `l` contient 0 ou 1 élément. Il s'agit cependant d'un warning et pas d'une erreur, ce qui veut dire que OCaml exécute quand même l'instruction mais nous prévient d'un problème potentiel.

[100]:
```
deuxieme []  (* donne une exception *)
```

```
Exception: Match_failure ("[99]", 1, 17).
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Voici comment on peut considérer tous les cas possibles :

[101]:
```
exception NotDefined;;
let deuxieme l = match l with  (* renvoie le 2ème élément d'une liste  *)
  | [] -> raise NotDefined  (* cas où la liste est vide *)
  | [e] -> raise NotDefined  (* cas où la liste n'a qu'un élément **)
  | e1::e2::q -> e2;;  (* cas où la liste a au moins 2 éléments *)
deuxieme [4; 7; 3]
```

[101]:
```
exception NotDefined
```

[101]:
```
val deuxieme : 'a list -> 'a = <fun>
```

[101]:
```
- : int = 7
```

On peut aussi utiliser underscore (_) dans un `match` pour signifier “dans tous les cas”. Par exemple, on peut regrouper les deux premiers cas de `deuxieme` :

[102]:
```
let deuxieme l = match l with
| e1::e2::q -> e2
| _ -> raise NotDefined  (* dans tous les autres cas *)
```

[102]:
```
val deuxieme : 'a list -> 'a = <fun>
```

Comme un match considère les cas de haut en bas, il faut mettre `| _ ->` en dernier sinon l’autre cas ne serait jamais exécuté.
Remarque underscore est aussi utilisé lorsque l’on ne souhaite pas nommer une variable (comme en Python) :

[103]:
```
let p = (1, 2) in
let x, _ = p in   (* récupère seulement x *)
x
```

[103]:
```
- : int = 1
```

[104]:
```
for _=0 to 5 do  (* autre exemple : exécuter 6 fois un for sans utiliser de␣
 ↪variable*)
  ()
done
```

[104]:
```
- : unit = ()
```

Exercice : Écrire une fonction `split : 'a list -> 'a list * 'a list` pour séparer une liste en deux listes de même taille (à 1 près). Les éléments d’indices pairs seront dans la 1ère liste, les autres dans la 2ème liste. **Exercice** Écrire une fonction `concat : 'a list -> 'a list -> 'a list` renvoyant une liste composée des deux listes en argument, l’une après l’autre. **Remarque** La fonction de l’exercice précédent existe déjà avec l’opérateur `@` :

[105]:
```
[1; 2; 3] @ [4; 5] (* concatène les deux listes et en renvoie une nouvelle *)
```

```
[105]: - : int list = [1; 2; 3; 4; 5]
```

Il est possible de décomposer plusieurs valeurs dans le même `match`, en les mettant sous forme de tuple :

```
[106]: let rec egal l1 l2 = match l1, l2 with  (* teste si l1 = l2 *)
         | [], [] -> true
         | e::q, [] -> false  (* l1 est non vide et l2 est vide *)
         | [], e::q -> false  (* l1 est vide et l2 est non vide *)
         | e1::q1, e2::q2 -> e1 = e2 && egal q1 q2;;

[106]: val egal : 'a list -> 'a list -> bool = <fun>

[107]: egal [1; 2; 3] [1; 2; 3];;
       egal [1; 2; 3] [1; 2];;

[107]: - : bool = true

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

IV.3.3 when

`when` permet de mettre une condition sur un cas d'un pattern matching. Il peut remplacer un `if`. Par exemple :

```
[108]: let rec positifs l = match l with  (* renvoie la liste des éléments positifs de l *)
         | [] -> []
         | e::q when e > 0 -> e::positifs q  (* on rajoute e à la liste de retour s'il est␣
       ↪positif *)
         | e::q -> positifs q;;  (* sinon on ajoute seulement les éléments positifs de q *)

[108]: val positifs : int list -> int list = <fun>
```

IV.4 Accumulateur de liste

Écrivons une fonction récursive `rev` pour inverser les éléments d'une liste. La méthode classique utilise un **accumulateur** (qu'on avait déjà utilisé pour calculer les termes de la suite de Fibonacci).

```
[109]: let rec rev acc l = match l with  (* acc va servir à construire le résultat (la␣
       ↪liste à l'envers) *)
         | [] -> acc
         | e::q -> rev (e::acc) q;;
       rev [] [1; 2; 3]  (* test *)

[109]: val rev : 'a list -> 'a list -> 'a list = <fun>

[109]: - : int list = [3; 2; 1]
```

Dans l'appel `rev [] l`, le premier élément `x` de `l` est celui qui est ajouté en premier à l'accumulateur et, comme tous les éléments suivants de `l` sont ajoutés au début de l'accumulateur (donc avant `x`), `x` se retrouve à la fin de l'accumulateur quand `rev` arrive sur le cas de base.
Le premier élément `x` se retrouve donc en dernier de l'accumulateur : on a bien inversé les éléments de `l`.

V Boucles

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

V.1 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
```

[110]:
```
for i=0 to 5 do
    print_int i
done;
print_newline()
```

```
012345
```

[110]:
```
- : 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 en 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" :

[111]:
```
for i=5 downto 0 do
    print_int i
done;
print_newline()
```

```
543210
```

[111]:
```
- : unit = ()
```

V.2 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` :

[112]:
```
let pgcd a b =
    let q = ref a in
    let r = ref b in
    while !r <> 0 do
        let reste = !q mod !r in
        q := !r;
        r := reste
    done;
    !q;;
pgcd 30 12;;
```

[112]:
```
val pgcd : int -> int -> int = <fun>
```

[112]:
```
- : int = 6
```

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

[113]:
```
let rec pgcd a b =
    if b = 0 then a
    else pgcd b (a mod b);;
pgcd 30 12;;
```

[113]:
```
val pgcd : int -> int -> int = <fun>
```

[113]:
```
- : int = 6
```

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

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

Écrire une fonction `temps_vol` ayant a en argument et renvoyant le premier indice n tel que $s_0 = a$ et $s_n = 1$. **Exercice (TP 2)** : l'écrire en récursif

V.3 Exceptions

Les exceptions sont déclenchées lorsque le programme rencontre un problème :

[114]:
```
1/0
```

```
Exception: Division_by_zero.
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

L'exception ci-dessus est `Division_by_zero`, déclenchée lorsque l'on divise par 0.
Il est possible de spécifier le comportement à adopter lors d'une exception :

[115]:
```
try 1/0
with Division_by_zero -> 0
```

[115]:
```
- : int = 0
```

OCaml exécute l'instruction dans le `try` (`1/0` ici).
Si cette instruction ne déclenche pas l'exception `Division_by_zero`, la valeur de l'instruction est renvoyée.
Sinon, C'est l'instruction dans le `with` qui est exécutée et renvoyée. **Exercice** Écrire une fonction `quotient : int -> int -> int` qui renvoie le quotient `a / b` si `b` est non-nul, et `max_int` sinon. Un autre exemple : `assert` est une fonction qui vérifie que son argument est `true` et déclenche une exception si ce n'est pas le cas. Ceci peut servir à mettre des tests dans le code.

[116]:
```
assert (1 = 2)
```

```
Exception: Assert_failure ("[116]", 1, 0).
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Il est possible de définir sa propre exception :

[117]:
```
exception Overflow;;  (* définition d'une exception Overflow qui affiche une chaîne␣
 ↪de caractères *)
let add1 n =
    if n = max_int then raise Overflow (* pour éviter un dépassement d'entier ù) *)
    else n + 1;;
add1 42;;
add1 max_int;;
```

[117]:
```
exception Overflow
```

[117]:
```
val add1 : int -> int = <fun>
```

[117]:
```
- : int = 43
```

```
Exception: Overflow.
Raised at file "[117]", line 4, characters 30-38
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

`failwith` permet de lancer une exception `Failure` (déjà définie en OCaml), avec un message d'erreur. `Failure` possède un argument (de type `string`, chaîne de caractères) permettant de donner des informations sur l'erreur :

[118]:
```
try failwith "Détail sur l'erreur"
with Failure e -> print_string e; print_newline ()
```

```
Détail sur l'erreur
```

[118]:
```
- : unit = ()
```

Voici comment sont définies `failwith` et `Failure` en OCaml:

[119]:
```
exception Failure of string;;  (* Failure possède un argument qui est de type string␣
 ↪*)
let failwith msg = raise (Failure msg);;
```

[119]:
```
exception Failure of string
```

[119]:
```
val failwith : string -> 'a = <fun>
```

Les exceptions peuvent aussi permettre de sortir d'une boucle (comme un `break` de Python). Même si c'est plutôt considéré comme une mauvaise pratique de programmation (car peut rendre le code plus compliqué), il y a certains cas où c'est justifié. **Exercice** : en utilisant une exception, écrire une fonction premier qui s'arrête dès qu'on a trouvé un diviseur.

[120]:
```
exception FoundDivisor;;
let premier n =
    try
        for i=2 to n/2 do  (* un diviseur de n est forcément inférieur à n/2 *)
            if n mod i = 0 then raise FoundDivisor
        done;
        true (* renvoyer true si on a pas trouvé de diviseur *)
    with FoundDivisor -> false
```

[120]:
```
exception FoundDivisor
```

[120]:
```
val premier : int -> bool = <fun>
```

VI Tableaux

Un tableau (`array`) se définit avec `[|...; ...; ...|]` :

[121]:
```
let a = [|1; 2; 3|]  (* tableau contenant 1, 2, et 3 *)
```

[121]:
```
val a : int array = [|1; 2; 3|]
```

Le type est `int array` : tableau d'entiers. On peut accéder à l'élément d'indice `i` d'un tableau avec `a.(i)` :

[122]:
```
a.(0)  (* 1er élément de a *)
```

[122]:
```
- : int = 1
```

Attention : Les indices d'un tableau de taille n vont de 0 à $n-1$. On obtient une erreur "index out of bounds" si on dépasse :

[123]:
```
let a = [|1; 2; 3|] in
a.(4)  (* a.(4) n'existe pas *)
```

```
Exception: Invalid_argument "index out of bounds".
Raised by primitive operation at unknown location
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

On peut modifier un élément avec `a.(i) <- ...` :

[124]:
```
a.(0) <- 5;
a  (* on a bien modifié a *)
```

[124]:
```
- : int array = [|5; 2; 3|]
```

Un array est donc **mutable**, contrairement aux listes.
Par contre la taille d'un tableau ne peut pas être modifié, une fois qu'on l'a crée.

[125]:
```
Array.length a  (* taille de a *)
```

[125]:
```
- : int = 3
```

Il est possible de créer un tableau de taille arbitraire en écrivant :

[126]:
```
Array.make 10 0;;  (* crée un tableau contenant 10 fois l'élement 0 *)
```

[126]:
```
- : int array = [|0; 0; 0; 0; 0; 0; 0; 0; 0; 0|]
```

Bien sûr, on peut remplir le tableau avec une autre valeur initiale :

[127]:
```
Array.make 7 1.23;;  (* crée un tableau de 7 éléments contenant 1.23 *)
```

[127]:
```
- : float array = [|1.23; 1.23; 1.23; 1.23; 1.23; 1.23; 1.23|]
```

Les tableaux sont passés par référence aux fonctions : si une fonction modifie un `array` en argument, elle le modifie aussi à l'extérieur de la fonction.
Exemple :

[128]:
```
let f a =  (* fonction qui met 42 dans la 1ere case du tableau a *)
  a.(0) <- 42 in
let t = [|0; -3; 4|] in
f t;
t  (* la modification de t à l'intérieur de f a modifié le tableau à l'extérieur *)
```

[128]:
```
- : int array = [|42; -3; 4|]
```

Exercice : Écrire une fonction pour calculer le maximum d'un tableau d'entiers. **Exercice** : Écrire une fonction `swap : 'a array -> int -> int -> unit` telle que `swap a i j` échange les éléments `a.(i)` et `a.(j)`.

VI.1 Matrices

Une matrice est un tableau de tableaux, ou, de façon équivalente, un tableau à 2 dimensions (signifiant qu'il faut donner 2 indices pour accéder à un élément). Exemple :

[129]:
```
let m = [|[|1; 2|]; [|3; 4|]|]
```

[129]:
```
val m : int array array = [|[|1; 2|]; [|3; 4|]|]
```

Cette matrice `m` correspond à la représentation mathématique suivante :

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

On peut sauter des lignes pour plus de clarté :

[130]:
```
let m = [|
  [|1; 2|]; (* 1ère ligne de la matrice m *)
  [|3; 4|]  (* 2ème ligne *)
|]
```

[130]:
```
val m : int array array = [|[|1; 2|]; [|3; 4|]|]
```

On voit d'après la définition de `m` que `m` est un tableau dont les éléments sont des tableaux. Le premier sous-tableau de `m` (c'est-à-dire `[|1; 2|]`) correspod à la 1ère ligne, par exemple.

[131]:
```
m.(0)  (* 2ème ligne de m *)
```

[131]:
```
- : int array = [|1; 2|]
```

On peut accéder à l'élément sur la ligne `i`, colonne `j` avec `m.(i).(j)` :

[132]:
```
m.(1).(0)  (* élément sur la ligne 1, colonne 0 de m *)
```

[132]:
```
- : int = 3
```

Exercice Comment connaitre le nombre de lignes d'une matrice `m`? Et son nombre de colonnes ? **Exercice** Écrire une fonction pour transposer une matrice `m`, c'est à dire obtenir `tm` telle que `tm.(i).(j) = m.(j).(i)`.
On pourra modifier `m` en place (sans créer de nouvelle matrice). Pour créer une matrice, on peut utiliser `Array.make_matrix` :

[133]:
```
Array.make_matrix 4 3 0  (* crée une matrice de 4 lignes et 3 colonnes remplie de 0␣
↪*)
```

[133]:
```
- : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

Attention On pourrait avoir envie de créer une matrice comme ci-dessous. Pourquoi est-ce en réalité une très mauvaise idée ?

[134]:
```
Array.make 4 (Array.make 3 0)
```

[134]:
```
- : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

VI.2 Chaîne de caractères (string)

Une chaîne de caractères permet de contenir du texte :

[135]:
```
let s = "Hello World"
```

[135]:
```
val s : string = "Hello World"
```

[136]:
```
s.[1]  (* le caractère à la position 1 *)
```

[136]:
```
- : char = 'e'
```

Un caractère (char) correspond à un symbole (lettre, chiffre, ponctuation…).
On ne peut pas modifier une string : ni ajouter un caractère ni en modifier un.

[137]:
```
s.[0] <- 'a'  (* ne marche pas : un string est persistant *)
```

```
File "[137]", line 1, characters 0-12:
1 | s.[0] <- 'a'  (* ne marche pas : un string est persistant *)
    ^^^^^^^^^^^^
Alert deprecated: Stdlib.String.set
Use Bytes.set instead.
```

```
File "[137]", line 1, characters 0-1:
1 | s.[0] <- 'a'  (* ne marche pas : un string est persistant *)
    ^
Error: This expression has type string but an expression was expected of type
         bytes
```

VII Résumé des structures de données d'OCaml

	tuple	list	array	string
exemple	`let t = (1, true, 3.14)`	`let l = [1; 2; 3]`	`let a = [|1; 2; 3|]`	`let s = "Hello"`
taille	taille connue à la création	utiliser une fonction récursive (ou `List.length`)	`Array.length`	`String.length`
décomposer	`let a, b, c = t`	`let e::q = l` ou `match l with| [] -> ... | e::q -> ...`	impossible	impossible
obtenir ième élément	décomposer le tuple	fonction récursive	`a.(i)`	`s.[i]`
modifier élément	impossible	impossible	`a.(i) <- ...`	impossible
ajouter élément	impossible	`e::l` (renvoie une nouvelle liste)	impossible	impossible
mutable	non	non	oui	non

VIII Types construits

En OCaml, on peut définir nos propres types.

VIII.1 Type somme (variant)

On peut définir un **type somme (variant)** en énumérant tous les cas possibles pour avoir ce type. Ceci revient à prendre l'union de plusieurs types. Chaque cas doit être identifié par un nom appelé **constructeur**, qui doit commencer par une majuscule (il est interdit de commencer un nom de variable avec une majuscule, pour ne pas confondre) :

[138]:
```
type matiere = Math | Physique | Info
```

[138]:
```
type matiere = Math | Physique | Info
```

On vient de définir un **type** `matiere` qui contient trois **valeurs** Math, Physique, Info.

[139]:
```
Math;;  (* c'est une valeur de type matiere *)
let a = Info;; (* qu'on peut stocker dans une variable *)
```

[139]:
```
- : matiere = Math
```

[139]:
```
val a : matiere = Info
```

Pour traiter une variable de ce type, on utilise un `match` :

[140]:
```
let heures m = match m with
  | Math -> 12.
  | Physique -> 6.5
  | Info -> 4.;;
heures Info
```

[140]:
```
val heures : matiere -> float = <fun>
```

[140]:
```
- : float = 4.
```

Les constructeurs utilisés ci-dessus sont des constantes, mais il est possible d'ajouter un paramètre avec `of ...` :

[141]:
```
type zbarre = Infini | MoinsInfini | Entier of int
```

[141]:
```
type zbarre = Infini | MoinsInfini | Entier of int
```

Dans cet exemple, `zbarre` est censé représenter $\overline{\mathbb{Z}} = \mathbb{Z} \cup \{-\infty, \infty\}$. `Entier` est un constructeur qui dépend d'un paramètre entier. Pour obtenir une valeur à partir du constructeur `Entier`, on l'utilise comme une fonction :

[142]:
```
Entier 10  (* valeur de type zbarre *)
```

[142]:
```
- : zbarre = Entier 10
```

Écrivons une fonction pour augmenter une valeur de type `zbarre` de 1 :

[143]:
```
let add1 n = match n with
  | Infini -> Infini
  | MoinsInfini -> MoinsInfini
  | Entier i -> Entier (i + 1);;
add1 (Entier 10);;
add1 Infini
```

[143]:
```
val add1 : zbarre -> zbarre = <fun>
```

[143]:
```
- : zbarre = Entier 11
```

[143]:
```
- : zbarre = Infini
```

Écrivons une fonction pour additionner deux valeurs de type `zbarre`. Pour pouvoir utiliser un opérateur infixe, on peut définir `(+!)` (liste de tous les symboles autorisés comme opérateur infixe):

[144]:
```
let (+!) n1 n2 = match n1, n2 with
| Infini, Infini -> Infini
| MoinsInfini, MoinsInfini -> MoinsInfini
| Infini, Entier(_) | Entier(_), Infini  -> Infini
| MoinsInfini, Entier(_) | Entier(_), MoinsInfini  -> MoinsInfini
| _ -> failwith "Indetermine"
```

[144]:
```
val ( +! ) : zbarre -> zbarre -> zbarre = <fun>
```

[145]:
```
Infini +! Infini;;
Infini +! MoinsInfini
```

[145]:
```
- : zbarre = Infini
```

```
Exception: Failure "Indetermine".
Raised at file "[119]", line 3, characters 25-38
Called from file "toplevel/toploop.ml", line 208, characters 17-27
```

Exercice : Définir `-!`, `*!`, `/!` pour des valeurs de type `zbarre`. Définir aussi `~-` qui permet d'avoir d'appliquer - sur un seul élément (opérateur unaire). Il est possible d'utiliser `'a` (n'importe quel type)

pour un paramètre de constructeur, à condition de le mettre dans le `type` (on appelle alors ceci un **type polymorphe**). Par exemple, on pourrait redéfinir `list` :

[146]:
```
type 'a liste = Vide | Cons of 'a * 'a liste
```

[146]:
```
type 'a liste = Vide | Cons of 'a * 'a liste
```

`Cons` est alors un constructeur ayant comme paramètre un couple : premier élément de la liste et reste de la liste. **Remarque** : `liste` est un **type récursif** (on utilise le type `liste` dans la définition de `liste`)

[147]:
```
let l = Cons(1, Cons(2, Cons(3, Vide))) (* équivalent de 1::2::3::[]) *)
```

[147]:
```
val l : int liste = Cons (1, Cons (2, Cons (3, Vide)))
```

[148]:
```
let rec appartient e l = match l with (* teste si e appartient à l *)
  | Vide -> false
  | Cons(x, q) -> x = e || appartient e q
```

[148]:
```
val appartient : 'a -> 'a liste -> bool = <fun>
```

Exercice : 1. Écrire un type `number` qui représente des nombres soient entiers (`int`), soit flottants (`float`). 2. Écrire des opérations (addition…) sur ce type. L'addition d'un flottant avec un entier donnera un flottant. 3. Écrire une fonction pour sommer les éléments d'une liste de `number`.

VIII.2 Type enregistrement (record)

Alors qu'un type somme fait une disjonction (un "ou") de plusieurs types, un **type enregistrement (record)** permet d'avoir plusieurs types simultanément (un "et" de plusieurs types).

[149]:
```
type fraction = {num: int; den: int};; (* fraction composée d'un numérateur ET
 ↪dénominateur *)
let x = {num=3; den=4};; (* x représente la fraction 1/4 *)
x.den  (* obtient la valeur du champ den de x *)
```

[149]:
```
type fraction = { num : int; den : int; }
```

[149]:
```
val x : fraction = {num = 3; den = 4}
```

[149]:
```
- : int = 4
```

`fraction` est un type très proche de `int*int`. La différence principale est que les composantes sont nommées dans un enregistrement mais pas dans un tuple.
Nous pouvons multiplier deux fractions :

[150]:
```
let mult x1 x2 =
  {num=x1.num*x2.num; den=x1.den*x2.den};;
  mult x x
```

[150]:
```
val mult : fraction -> fraction -> fraction = <fun>
```

[150]:
```
- : fraction = {num = 9; den = 16}
```

Exercice : 1. Écrire une fonction pour additionner deux fractions. 2. Écrire une fonction pour simplifier une fraction. L'intérêt d'utiliser des fractions plutôt que des float est de faire des calculs exacts et non pas approchés. **Exercice** : 1. Définir deux types enregistrements pour représenter un nombre complexe

sous forme algébrique et sous forme polaire. 2. Écrire une fonction pour convertir un nombre complexe de polaire à algébrique.

VIII.2.1 Mot-clé mutable

Par défaut, un type enregistrement n'est pas mutable (on ne peut pas modifier ses éléments). Cependant, il est possible d'ajouter un mot-clé mutable sur un attribut. On utilise alors `<-` pour modifier l'attribut. Par exemple, on pourrait redéfinir une référence comme un type enregistrement avec un seul champ mutable :

[151]:
```
type 'a ref = {mutable v: 'a}
```

[151]:
```
type 'a ref = { mutable v : 'a; }
```

[152]:
```
let a = {v = 2}  (* équivalent de let a = ref 2 *)
```

[152]:
```
val a : int ref = {v = 2}
```

[153]:
```
a.v  (* équivalent de !a *)
```

[153]:
```
- : int = 2
```

[154]:
```
a.v <- 5  (* équivalent de a := 5 *)
```

[154]:
```
- : unit = ()
```