cours1

November 3, 2020

1 Programmation fonctionnelle

1.1 Semaine 1 : Programmer avec des fonctions

Etienne Lozes - Université Nice Sophia Antipolis - 2019

1.2 Calculer et programmer

1.2.1 Les deux grands *modèles de calcul* historiques sont

- les **machines de Turing** (1936)
- le **lambda calcul** (1936 aussi), inventé par **Alonzo Church**, le directeur de thèse de Turing

1.2.2 Ces deux modèles correspondent à deux grandes familles de langages de programmation

- la programmation **impérative** (~ machines de Turing). On décrit **comment** calculer par des suites d'opérations
- la programmation **fonctionnelle** (~ le lambda calcul). On décrit **quoi** calculer à l'aide de fonctions.

Remarque: la programmation fonctionnelle, en ce sens, est **déclarative**. Un autre exemple de programmation déclarative est la programmation logique (Prolog).

1.3 Où rencontre-t-on la programmation fonctionnelle?

Dans les langages fonctionnels, évidemment, mais pas seulement… - dans des **langages "impératifs"** (ex: Python), par exemple pour trier une liste avec un critère de tri particulier - dans les "interfaces utilisateurs" (ex: Javascript), pour **gérer les évènements** souris, clavier, etc - dans les langages "parallèles" (ex: Erlang), pour gérer des **processus** (*thread*) - dans des middleware dédiés au big data (**MapReduce**) - dans les **applications critiques**, la programmation fonctionnelle ayant des liens étroits avec la logique et la **théorie de la démonstration** (COQ)

1.3.1 Plusieurs familles de langages fonctionnels

Par ordre chronologique: 1. **LISP** (Scheme, Racket, …) - **importance historique**, introduction de concepts fondamentaux comme la **liaison tardive** ou les **continuations** 2. **ML** (SML, Caml, Microsoft F# pour .NET , …) - a introduit l'**inférence de type** qui permet de s'affranchir des anotations de type (cf Java) sans sacrifier la sûreté (cf Python) 3. **Haskell** : - a promu l'**évaluation paresseuse** et les **monades**, et a poussé la philosophie déclarative à l'extrême

Les *lambdas* ont fini par s'imposer dans les langages impératifs - en python 3: `lam x : x + 1` - en Java (à partir de la version 1.8) : `(x) -> x+1` - en C++ (à partir de la version 11) : `[] (int i) {return i+1;}`

1.4 Le langage OCaml

1.4.1 Un langage de la famille ML

- le premier langage de cette famille, **standard ML** (SML), est issu des travaux de recherche de **Robin Milner**
- SML introduit l'**inférence de type** automatique ainsi que le **pattern-matching** (*filtrage par motif* en québecois).
- l'INRIA s'empare du concept et développe Caml dans les années 1980

1.4.2 De Caml Light à OCaml

- Dans les années 1990, le compilateur devient de plus en plus performant, et la gestion mémoire est simplifiée grace à un **garbage collector** (GC en geek, *ramasse-miette* en québécois) très efficace, grace aux travaux de **Xavier Leroy** et **Damien Doligez**
- Il est adopté comme langage d'enseignement en classes préparatoires
- Le langage s'enrichit d'un système de **modules**, d'**objet**, et de **labels** donnant naissance à OCaml au début des années 2000

1.4.3 Aujourd'hui : l'âge OCaml Pro

- Durant les années qui suivent, OCaml trouve peu à peu sa place dans l'industrie sur certains créneaux, en particulier la **sûreté**, notamment avionique (Astrée, Airbus), la **sécurité** (Cryptosense, TrustInSoft), la **finance** (Jane Street Capital, Lexify), la **programmation web** (Ocsigen), le **cloud** (MirageOS), etc
- Il est aussi adopté largement hors de France pour l'enseignement de la programmation fonctionnelle (Cornell, Harvard, **MIT**, Cambridge, Pise, Bologne, Innsbruck, …), concurrençant le très américain **Haskell**
- la société **OCaml Pro** est créée en 2011 pour vivifier la communauté d'utilisateurs

---

1.5 Ressources sur OCaml

1.5.1 La page officielle www.ocaml.org

- **OCaml de base** (compilateur, toplevel, profiler, …)
- **OPAM**, le gestionnaire de paquets (recommandé!)

1.5.2 La documentation en ligne

- le manuel de référence de l'utilisateur
- le livre Developing Applications With Objective Caml
- un cours plus avancé pour comprendre un peu mieux comment Caml fonctionne

---

1.6 Premiers pas : le toplevel

1.6.1 Rappel : la boucle REPL

Comme avec Python, le toplevel d'Ocaml est une boucle REPL (*read-evaluate-print loop*) qui va vous permettre de tester du code et de l'exécuter immédiatement.

1.6.2 Comment lancer un toplevel?

1. depuis un navigateur web, sans même avoir installé OCaml, grâce à **TryOCaml**

https://try.ocamlpro.com .

2. depuis un terminal, il suffit de taper la commande `ocaml`.
   - pas toujours très pratique… mais c'est le toplevel officiel
   - devient plus pratique si on le lance depuis *Emacs*
3. en installant un toplevel non-officiel mais plus populaire
   - ocaml-top fait l'indentation automatique et la coloration syntaxique
   - utop offre la complétion automatique

1.6.3 Quelques remarques sur le toplevel

- Avant d'afficher la valeur calculée, le toplevel affiche son **type** (`int`, `string`, `float`, …)
- On peut écrire sur plusieurs lignes, c'est `;;` qui marque la fin de la saisie
- La commande `#quit` interrompt le toplevel
  - attention! cela ne marche qu'avec le toplevel
  - on peut aussi utiliser l'expression `exit 0`, valable dans tout programme

---

1.7 Les entiers

En Ocaml, **les entiers sont limités** (ils occupent un nombre fixe de bits en mémoire). Il y a donc un plus grand et un plus petit entier.

[1]:
```
max_int
```

[1]:
```
- : int = 4611686018427387903
```

[2]:
```
max_int + 1
```

[2]:
```
- : int = -4611686018427387904
```

[3]:
```
min_int = max_int + 1
```

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

L'exponentiation n'existe pas pour les entiers (elle est réservée aux flottants). :-(

[4]:
```
2 ** 50
```

```
File "[4]", line 1, characters 0-1:
1 | 2 ** 50
    ^
Error: This expression has type int but an expression was expected of type
         float
```

[5]:
```
2.0 ** 50.0
```

[5]:
```
- : float = 1125899906842624.
```

1.8 Les flottants

Ocaml fait la distinction entre les entiers et les flottants.

[6]:
```
2
```

[6]:
```
- : int = 2
```

[7]:
```
2.0
```

[7]:
```
- : float = 2.
```

Le `0` derrière le `.` est facultatif.

Les opérations sur les flottants sont `+.`, `*.`, `-.`, `/.`, et `**`.

Elles sont distinctes des opérations sur les entiers

[8]:
```
1 + 2 * 3 / 4
```

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

```
[9]: 1. +. 2. *. 3. /. 4.
```

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

Il n'y a aucune opération arithmétique qui puisse associer un entier et un flottant

```
[10]: 2 +. 3. (* erreur de débutant classique, on a oublié le . après le 2 *)
```

```
File "[10]", line 1, characters 0-1:
1 | 2 +. 3. (* erreur de débutant classique, on a oublié le . après le 2 *)
    ^
Error: This expression has type int but an expression was expected of type
         float
```

```
[11]: 2. + 3.  (* autre classique, on a oublié le . après le + *)
```

```
File "[11]", line 1, characters 0-2:
1 | 2. + 3.  (* autre classique, on a oublié le . après le + *)
    ^^
Error: This expression has type float but an expression was expected of type
         int
```

Par contre on peut explicitement convertir un entier en flottant et vice-versa

```
[12]: int_of_float(2. ** 50.)
```

```
[12]: - : int = 1125899906842624
```

```
[13]: float_of_int(max_int) +. 1.
```

```
[13]: - : float = 4.6116860184273879e+18
```

1.9 Les chaînes de caractères

[14]:
```
"les chaines peuvent être
sur plusieurs lignes"
```

[14]:
```
- : string = "les chaines peuvent être \nsur plusieurs lignes"
```

[81]:
```
let s= {|on peut se passer de "double quote" comme délimiteur|}
```

[81]:
```
val s : string = "on peut se passer de \"double quote\" comme délimiteur"
```

L'opérateur de contanétation de chaînes est `^`

[16]:
```
"a" ^ "b"
```

[16]:
```
- : string = "ab"
```

Pour connaitre la longueur d'une chaîne, on appele la fonction `length` du module `String` (attention au *S* majuscule)

[17]:
```
String.length("abcdefghijklmnopqrstuvwxyz")
```

[17]:
```
- : int = 26
```

Pour acceder au caractère d'indice i de s, on écrit `s.[i]` (au lieu de `s[i]` en Python).

[18]:
```
"abcd".[1]
```

[18]:
```
- : char = 'b'
```

Notez qu'un caractère individuel s'écrit avec un *quote* `'` simple et a le type `char` et non `string`. On peut passer du code ASCII au caractère et vice versa avec les fonctions `Char.code` et `Char.chr`

[19]:
```
Char.code('a'), Char.chr(97)
```

[19]:
```
- : int * char = (97, 'a')
```

1.10 Les booléens

[20]:
```
1 = 1
```

[20]:
```
- : bool = true
```

[21]:
```
0 = 1
```

[21]:
```
- : bool = false
```

[22]:
```
1 / 0 > 0
```

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

Les booléens se notent `true` et `false`; le *et* logique se note `&&` et le *ou* `||`; ce sont des opérateurs paresseux, comme en Python.

[83]:
```
3 > 2 && (2 <= 2 || 1 / 0 > 0)
```

[83]:
```
- : bool = true
```

L'expression ci-dessus ne provoque pas d'erreur de division par 0; elle est rigoureusement équivalente à

[24]:
```
if not (3 > 2)
then false
else if 2 <= 2
     then true
     else 1 / 0 > 0
```

[24]:
```
- : bool = true
```

1.11 Le typage statique

Contrairement à Python, OCaml ne permet aucune confusion entre entiers, flottants, booléens, ou chaînes de caractères

[25]:
```
1 = true
```

```
File "[25]", line 1, characters 4-8:
1 | 1 = true
        ^^^^
```

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

[86]:
```
1 + "2"
```

```
File "[86]", line 1, characters 5-8:
1 |  1 + "2"
         ^^^
Error: This expression has type string but an expression was expected of type
         int
```

RÈGLE D'OR DU TYPAGE STATIQUE

> **Toute expression a un type**, que le toplevel, et plus généralement le compilateur, doit pouvoir déterminer **avant de lancer le calcul**.

1.12 Les expressions

En OCaml, tout est expression. Le `if..then..else` ne déroge pas à la règle: Ocaml refuse que le type de l'expression de la branche `else` soit différent de celui de la branche `then`: on ne saurait pas avant de lancer le calcul quel sera le type du résultat de l'expression conditionnelle!

[27]:
```
if Random.int(2) = 0 then "un" else 0
```

```
File "[27]", line 1, characters 36-37:
1 | if Random.int(2) = 0 then "un" else 0
                                        ^
Error: This expression has type int but an expression was expected of type
         string
```

[28]:
```
if Random.int(2) = 0 then "un" else "zéro"
```

[28]:
```
- : string = "un"
```

1.12.1 Les instructions sont des expressions comme les autres

Leur type est `unit`, l'équivalent du `None` de Python. Le type `unit` contient une seule valeur, notée `()`. On peut voir `()` comme *le* tuple de dimension 0.

[29]:
```
print_int(42)
(* affiche 42 avant de rendre la main. La valeur calculée est (), et surtout␣
 ↪pas 42! *)
```

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

Autrement dit, une instruction est une expression dont la valeur, `()`, peut être connue à l'avance, à ceci près que son évaluation ne termine pas forcément. *Par abus de langage* on considère parfois qu'une instruction n'a pas de valeur.

1.13 "C'est pénible OCaml, monsieur!"

Vous commencez à vous en rendre compte, le typage est une notion très importante en OCaml.

Quand on vient d'un langage typé dynamiquement comme Python (ou LISP, pour rester dans les langages fonctionnels), on trouve ça parfois une mauvaise idée de mettre autant des bâtons dans les roues aux programmeurs.

1.13.1 Oui, mais...

ces batons ont parfois du bon pour éviter des erreurs! Si Java tient encore le haut du classement des langages les plus utilisés, devant Python, c'est peut-être parce qu'il est typé statiquement (en partie, tout le moins), ce qui entraine parfois des lourdeurs bien pire qu'en OCaml... donc cessez de râler et accrochez-vous!

1.14 À propos de l'égalité

Vous avez sans doute noté que le test d'égalité se note `=` avec **un seul** `=`. - l'affectation existe, elle se note `:=` ou `<-` selon les cas , mais patience, c'est encore un peu tôt pour en parler... - `==` existe, mais correspond à l'**égalité physique** que l'on notait `is` en Python - si `a==b`, alors `a` et `b` désignent le même objet en mémoire, donc en particulier `a=b`, mais on n'a pas forcément la réciproque... méfiez-vous!

[30]:
```
"a" ^ "b" == "ab" (* FAUX! *)
```

[30]:
```
- : bool = false
```

En effet, l'opération de concaténation alloue une nouvelle adresse mémoire pour stocker le résultat de la concaténation. On a donc **deux** chaînes "ab" en mémoire à des adresses différentes: l'une à l'adresse allouée par la concaténation, l'autre à l'adresse allouée par le toplevel quand il a lu `"ab"` à droite du `==`.

Pour la même raison, évitez le `!=`, qui est la négation de `==`. La négation de `=` se note `<>`.

[31]:
```
0 <> 1
```

[31]:
```
- : bool = true
```

1.15 Les définitions

Une définition permet de donner un nom à une expression en vue de l'utiliser par la suite. C'est l'équivalent du `def` de Python, en quelque sorte... mais sans être restreint aux fonctions. Il y a deux sortes de définitions. - Les **définitions globales** sont valables pour toute la suite de la session du toplevel (ou toute la suite du programme). Elles s'écrivent `let ... = ...` - Les **définitions locales** s'écrivent `let ... = ... in ...` . La *portée* de la définition est restreinte à l'expression à droite du `in`. - le nom de la variable ainsi définie doit nécesssairement commencer par une **minuscule** ou un **underscore** (_). Les caractères suivants peuvent être des chiffres, des lettres majuscules ou minuscules, ou l'un des caractères _ ou '

[32]:
```
let pi = 4. *. atan(1.0) (* la définition de pi reste valable pour toute la␣
↪suite *)
```

[32]:
```
val pi : float = 3.14159265358979312
```

[33]:
```
let e = exp(1.) in e +. pi  (* la définition de e est restreinte à cette ligne␣
↪*)
```

[33]:
```
- : float = 5.85987448204883776
```

[34]:
```
let _MaDef' = e *. pi
```

```
File "[34]", line 1, characters 14-15:
1 | let _MaDef' = e *. pi
                  ^
Error: Unbound value e
```

[35]:
```
let PI = 3.14
```

```
File "[35]", line 1, characters 4-6:
1 | let PI = 3.14
        ^^
Error: Unbound constructor PI
```

1.16 Portée d'une définition

Une définition peut en masquer une autre, temporairement si c'est une définition locale, ou définitivement si c'est une définition globale.

[36]:
```
let pi = 4. *. atan(1.)
```

[36]:
```
val pi : float = 3.14159265358979312
```

[37]:
```
let pi2 =
    let pi = 3.14 (* cette définition masque l'autre à l'intérieur du bloc in *)
    in  pi *. pi
```

[37]:
```
val pi2 : float = 9.8596
```

à la fin du bloc `in`, la définition `pi = 3.14` a disparu

[38]:
```
pi, pi *. pi
```

[38]:
```
- : float * float = (3.14159265358979312, 9.86960440108935799)
```

[39]:
```
let pi = 3.14 (* masque définitivement la définition pi = 4. *. atan(1.) *)
```

[39]:
```
val pi : float = 3.14
```

RÈGLE D'OR DES DÉFINITIONS

En cas de définitions multiples d'une même variable, c'est la dernière définition encore valable qui prévaut.

1.17 Les définitions de fonctions

La syntaxe pour définir les fonctions est quasiment la même que pour les variables; il faut seulement faire attention à mettre le mot clé `rec` après `let` lorsque c'est une fonction récursive.

[40]:
```
let add1(n) = n + 1
let abs(x) = if x > 0. then x else -. x
let rec fact(n) = if n = 0 then 1 else n * fact(n - 1)
```

[40]:
```
val add1 : int -> int = <fun>
```

[40]:
```
val abs : float -> float = <fun>
```

[40]:
```
val fact : int -> int = <fun>
```

[41]:
```
add1(1), abs(-. 6.), fact(13)
```

[41]:
```
- : int * float * int = (2, 6., 6227020800)
```

Ce mot-clé `rec` est un peu pénible; si on l'oublie, Ocaml fait une erreur et suggère comment la corriger.

[42]:
```
let f(n) = if n = 0 then 1 else n * f(n-1)
```

```
File "[42]", line 1, characters 36-37:
1 | let f(n) = if n = 0 then 1 else n * f(n-1)
                                        ^

Error: Unbound value f
Hint: If this is a recursive definition,
you should add the 'rec' keyword on line 1
```

1.18 La raison d'être du rec

Pourquoi OCaml réclame-t-il donc ce mot-clé `rec` puisqu'il est capable de suggérer comment corriger?

Parce qu'il en a besoin pour être sûr de bien comprendre ce que vous voulez dire!

Un petit exemple permet de cerner le problème.

[43]:
```
let f(n) = 0
let f(n) = if n = 0 then 1 else n * f(n-1)
let res = f(0), f(6)
```

[43]:
```
val f : 'a -> int = <fun>
```

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

[43]:
```
val res : int * int = (1, 0)
```

Notez que `f(0)` renvoie `1` mais `f(6)` renvoie `0`! C'est normal: la deuxième définition de `f` se base sur la première, donc `6 * f(5)` est remplacé par `6 * 0`. Si on rajoute le mot-clé `rec`, la deuxième définition de `f` correspond bien à la définition de la factorielle.

[44]:
```
let f(n) = 0
let rec f(n) = if n = 0 then 1 else n * f(n-1)
let res = f(0), f(6)
```

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

```
val f : int -> int = <fun>
```

```
val res : int * int = (1, 720)
```

1.19 Les fonctions à plusieurs arguments

Pour l'instant, nous n'avons défini que des fonctions à un seul argument – on dit aussi des fonctions d'**arité** 1. Pour définir une fonction à plusieurs arguments, autrement dit une fonction d'arité $n \geq 0$ un entier quelconque, il est possible d'utiliser des virgules pour les séparer, comme on le ferait en Python.

[45]:
```
let moyenne(a, b) = (a +. b) /. 2.        (* une fonction d'arité 2 *)
```

```
val moyenne : float * float -> float = <fun>
```

`moyenne` est une fonction qui a un couple de flottants (`float * float`) associe un flottant. En math, on écrirait

$$moyenne : (\mathbb{R} \times \mathbb{R}) \to \mathbb{R}$$

avec des parenthèses, omises par Ocaml

[46]:
```
let derivee(f, x, h) = (f(x +. h) -. f(x)) /. h    (* une fonction d'arité 3 *)
```

```
val derivee : (float -> float) * float * float -> float = <fun>
```

`derivee` est une fonction qui a un triplet associe un flottant. Le premier élément de ce triplet est une fonction de type `float->float`, et les deux autres éléments du triplet sont des flottants

$$derivee \quad : \quad \left((\mathbb{R} \to \mathbb{R}) \times \mathbb{R}^2\right) \quad \to \quad \mathbb{R}$$

[47]:
```
let piece_truquee () = (* une fonction d'arité 0 *)
    (* renvoie "pile" avec probabilité 2/3 *)
    let x=  Random.int(3) = 0 then "face" else "pile"
```

[47]:
```
val piece_truquee : unit -> string = <fun>
```

la fonction `piece_truquee`, du point de vue d'OCaml, **n'est pas** une fonction sans argument! C'est une fonction qui prend un argument de type `unit` et qui renvoie une chaîne de caractères. Comme la seule valeur possible pour l'argument de type `unit` est `()`, on peut *moralement* la considérer comme une fonction sans argument... mais c'est un abus de langage!

$$piece\_truquee : \mathbf{1} \to \mathsf{STRING}$$

où **1** désigne un ensemble à un seul élément.

Tout compte fait, du point de vue d'OCaml, on a défini des fonctions à un seul argument en utilisant les tuples pour simuler plusieurs arguments...

RÈGLE D'OR SUR L'ARITÉ DES FONCTIONS

> Du point de vue d'OCaml, toutes les fonctions sont d'arité 1: elles prennent exactement un argument.

1.20 Les fonctions qui renvoient des fonctions

Sur l'exemple de la fonction `derivee`, nous avons vu que l'on pouvait passer une fonction en argument à une autre fonction.

En programmation fonctionnelle, **les fonctions sont des valeurs comme les autres**! Le résultat d'une fonction peut donc aussi être une fonction.

Les matheux utilisent souvent dans ce cas-là les mots *opérateur* ou *fonctionnelle*...

Voyons donc comment on peut définir l'*opérateur de dérivation* en OCaml grace à une définition locale

[48]:
```
let derivee(f, h) =
   let f'(x) = (f(x +. h) -. f(x)) /. h
   in f'
```

[48]:
```
val derivee : (float -> float) * float -> float -> float = <fun>
```

`derivee (f,h)` renvoie le résultat de l'évaluation de toute l'expression `let f'(x) = ... in f'`, autrement dit `derivee (f,h)` renvoie `f'`, la dérivée approchée de `f`!

Vous suivez?

Ignorons le type de la fonction `derivee` pour le moment, et voyons comment utiliser notre opérateur de dérivation.

[49]:
```
let g = derivee(sin, 0.00001) (* g est la dérivée approchée de la fonction␣
 ↪sinus *)
```

[49]:
```
val g : float -> float = <fun>
```

Le type de `g` nous assure que `g` est bien une fonction qui prend un flottant et renvoie un flottant. Vérifions s'il s'agit bien de la dérivée de sinus (autrement dit, cosinus).

[50]:
```
g(pi /. 2.)
```

[50]:
```
- : float = 0.000791326715265938565
```

[51]:
```
cos(pi /. 2.)
```

[51]:
```
- : float = 0.000796326710733263345
```

1.21 La véritable notation de l'appel de fonction en OCaml

Il est temps de dire la vérité: traditionnellement on écrit `f x` au lieu de `f(x)` en Ocaml. La notation `f(x)` permet simplement de *faire comme si* un espace séparait `f` de `x`, mais les programmeurs OCaml ne font pas ça. On va donc désormais écrire `f x` pour un appel de fonction et `let f x = ...` pour une définition de fonction.

[52]:
```
let inc n = (* <- notez l'absence de parenthèses! *)
    let f m = n + m (* <- ici aussi ! *)
    in f
```

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

[53]:
```
inc 3 (* la fonction qui incrémente de 3 *)
```

[53]:
```
- : int -> int = <fun>
```

Avez-vous remarqué le type de `inc`? On retrouve la même bizarerie que pour `derivee`: normal, il s'agit à nouveau d'une fonction dont le résultat est une fonction. Voyons cela de plus près cette fois-ci.

On pourrait s'attendre à ce qu'OCaml affiche quelque chose comme

```
val inc : int -> (int -> int) = <fun>
```

mais, *par convention*, il se permet d'omettre les parenthèses. La flêche `->` est en effet considérée en informatique, et dans certaines branche des mathématiques, comme associative à droite, autrement dit

a -> b -> c $\Leftrightarrow$ a -> (b -> c)

Le pendant de cette convention est qu'on peut aussi écrire

`f x y` au lieu de `(f x) y`.

[54]:
```
inc 3 4
```

[54]:
```
- : int = 7
```

On a écrit `inc 3 4`, mais on aurait pu écrire `(inc(3))(4)` comme en mathématique, ou `(inc 3) 4` si l'on n'a pas en tête le fait qu'OCaml suit la convention `f x y = (f x) y`… on s'y fait vite, courage!

1.22 Toutes les parenthèses ne sont pas bonnes à jeter

Cela va sans dire… mais cela va mieux en le disant, et en observant quelques exemples. Observons ce qui se passe lorsqu'on retire les parenthèses à

`cos(pi /. 2.)`.

[55]:
```
cos pi /. 2.
```

[55]:
```
- : float = -0.499999365863769751
```

Et oui, on a calculé $\frac{cos(\pi)}{2}$ et non $cos(\frac{pi}{2})$! Autrement dit, OCaml a interprété notre ligne comme étant `(cos pi) /. 2.`.

RÈGLE DE MISE ENTRE PARENTHÈSES DE L'ARGUMENT

> Lorsque l'argument `e` d'une fonction `f` est une expression composée (`e` n'est ni une constante explicite, ni un nom de variable), on devra écrire `f(e)` ou `f (e)`, et non `f e`.

[56]:
```
moyenne(1., 2.)  (* CORRECT *)
```

[56]:
```
- : float = 1.5
```

[57]:
```
moyenne 1., 2.
(* INCORRECT: OCaml comprend (moyenne 1.), 2. et râle car 1. n'est pas un␣
 ↪couple *)
```

```
File "[57]", line 1, characters 8-10:
1 | moyenne 1., 2.
            ^^
Error: This expression has type float but an expression was expected of type
         float * float
```

1.23 Les couples

Les couples (*pair* en anglais) se notent à l'aide d'une virgule, les parenthèses ne sont pas obligatoires.

[58]:
```
let c = 1,2
```

[58]:
```
val c : int * int = (1, 2)
```

On peut **déconstruire** un couple avec les fonctions `fst` et `snd`

[59]:
```
fst c
```

[59]:
```
- : int = 1
```

[60]:
```
snd c
```

[60]:
```
- : int = 2
```

On peut aussi utiliser la construction `let...=....` pour *filtrer* la valeur qui nous intéresse, un peu comme on ferait en Python

[61]:
```
let x,y = c in x
```

[61]:
```
- : int = 1
```

On peut donc facilement reprogrammer la fonction `fst`

[62]:
```
let fst c =
   let x,y = c in x
```

[62]:
```
val fst : 'a * 'b -> 'a = <fun>
```

Ou plus simplement

[63]:
```
let fst (x,y) = x
```

[63]:
```
val fst : 'a * 'b -> 'a = <fun>
```

OCaml propose les tuples d'arités quelconque, sauf d'arité 1.

[64]:
```
let quadruplet = "joe", "jack", "william", "averell"
```

[64]:
```
val quadruplet : string * string * string * string =
  ("joe", "jack", "william", "averell")
```

Par contre, on ne peut pas accèder par son indice au ième élément d'un tuple

[65]:
```
quadruplet[1]
```

```
File "[65]", line 1, characters 0-10:
1 | quadruplet[1]
    ^^^^^^^^^^
Error: This expression has type string * string * string * string
       This is not a function; it cannot be applied.
```

1.24 Fonctions à plusieurs paramètres : la curryfication

Nous avons vu comment écrire des fonctions à plusieurs paramètres en utilisant des tuples. Autant dire la vérité: les programmeurs OCaml ne font presque jamais comme ça! Les programmeurs OCaml utilisent la **curryfication**.

Malgré son nom un peu impressionnant, c'est une technique assez simple à comprendre: il s'agit de représenter une fonction $f : (A \times B) \to C$ d'arité 2 à l'aide d'une fonction $f' : A \to (B \to C)$.

La version *débutant* d'une fonction à plusieurs paramètres

[66]:
```
let f(s, i) = s.[i] (* NOTE : cette fonction renvoie le caractère d'indice i de
 ↪s *)
```

[66]:
```
val f : string * int -> char = <fun>
```

et la version *curryfiée* de cette même fonction, dans le style typique d'OCaml

[67]:
```
let f' s i = s.[i]
```

[67]:
```
val f' : string -> int -> char = <fun>
```

Autrement dit, au lieu de passer les deux arguments d'un coup en les groupant dans un couple, on les passe un par un: lorsqu'on a passé seulement le premier argument on obtient une fonction qui attend le second argument.

Il peut être intéressant d'**évaluer partiellement** `f'` en lui passant seulement son premier argument.

[68]:
```
let lettre = f' "abcdefghijklmnopqrstuvwxyz"
```

[68]:
```
val lettre : int -> char = <fun>
```

définit la fonction qui à un entier i associe la $i$ème lettre de l'alphabet

[69]:
```
lettre 0
```

[69]:
```
- : char = 'a'
```

La définition de `f'` précédente est rigoureusement équivalente à

[70]:
```
let f' s =
   let h i = s.[i]
   in h
```

[70]:
```
val f' : string -> int -> char = <fun>
```

1.25 L'opérateur de dérivation en version curryfiée

On vient de voir comment définir l'opérateur de dérivation

[71]:
```
let derivee(h,f,x) = (f(x +. h) -. f(x)) /. h
```

[71]:
```
val derivee : float * (float -> float) * float -> float = <fun>
```

Le programmeur Ocaml préfèra presque toujours une version curryfiée de cette définition.

[72]:
```
let derivee_curry h f x =  (f(x +. h) -. f(x)) /. h
```

[72]:
```
val derivee_curry : float -> (float -> float) -> float -> float = <fun>
```

Pour comprendre l'intérêt de la chose, voyons comment utiliser cette fonction `derivee_curry`. Comme on vient de le voir, on peut bien sûr écrire `derivee_curry 0.0001 sin (pi /. 2.)` avec des espaces, ce qui a pour effet de calculer une approximation de $sin'(\frac{\pi}{2})$ avec la précision $1/10000$. Mais on peut aussi **évaluer partiellement** la fonction `derivee_curry` et obtenir d'autres fonctions tout aussi intéressantes.

[73]:
```
let sin' = derivee_curry 0.0001 sin
```

[73]:
```
val sin' : float -> float = <fun>
```

Si l'on omet le point auquel on dérive, on obtient une fonction qui attend le dernier argument manquant `x` pour effectuer le calcul de la dérivée en `x`: ce n'est rien d'autre que la fonction dérivée sin', égale à cos.

Encore plus fort! si l'on omet de préciser la fonction `f` à `derivee_curry`, on obtient l'opérateur de dérivation parfois noté $\frac{d}{dx}$ en physique.

[74]:
```
let d_dx = derivee_curry 0.0001
```

[74]:
```
val d_dx : (float -> float) -> float -> float = <fun>
```

et l'on peut donner une définition plus élégante de sin'

[75]:
```
let sin' = d_dx sin
```

[75]:
```
val sin' : float -> float = <fun>
```

1.25.1 L'opérateur de composition

La composée $f \circ g$ est la fonction qui à x associe $f(g(x))$: on passe x en paramètre à la fonction g, puis on passe le résultat à la fonction f. On note aussi parfois $g; f$ au lieu de $f \circ g$. Sur un dessin:

Comment définirait-on l'opérateur $\circ$ en OCaml? Si l'on n'est pas avare de parenthèses, on pourra écrire

[76]:
```
let composee f g =
    let f_rond_g(x) = f(g(x)) in
    f_rond_g
```

[76]:
```
val composee : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

Le type

```
('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

de `composee` est un peu déroutant, démystifions-le avant de continuer davantage.

D'abord, ne tenez pas compte de ces apostrophes, elles signifient simplement que `a, b, c` sont des *inconnues* de type, on aura l'occasion d'en reparler…

Ensuite, comme on l'a vu, la flèche $\rightarrow$ est associative à droite. On peut donc réécrire le type de `composee` comme

$$(a \rightarrow b) \rightarrow \Big[(c \rightarrow a) \rightarrow (c \rightarrow b)\Big]$$

Autrement dit, composee attend en argument deux fonctions f et g telles que l'ensemble de départ de f soit aussi l'ensemble d'arrivée de g (le type a).

De plus, `composee` renvoie une fonction dont l'ensemble de départ est celui de `g` et l'ensemble d'arrivée est celui de `f`.

Ça tombe bien, c'est ce qu'on voulait!

[77]:
```
let composee f g =
    let f_rond_g(x) = f(g(x)) in
    f_rond_g
```

[77]:
```
val composee : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

1.25.2 L'opérateur de composition, seconde version

Regardez à nouveau le type de `composee` tel qu'OCaml l'écrivait:

```
val composee : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b
```

Après tout, pourquoi ne verrait-on pas `composee` comme une fonction à trois arguments, dont le troisième de type `c` serait le point x auquel on évalue la composée? Si l'on adopte ce point de vue, on obtient une définition beaucoup plus "stylée" de la composée, tout à fait dans l'esprit d'OCaml.

[78]:
```
let composee f g x = f (g x)
```

[78]:
```
val composee : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

1.26 Haskell Brooks Curry (1900-1982)

Ce logicien de la période "pré-informatique" a eu une influence remarquable sur la programmation fonctionnelle: le mot *curryfication* lui doit son nom, de même que trois langages de programmation: le célèbre Haskell (très connu), mais aussi Brook, et Curry. Pas mal pour une seule personne!

Mais surtout, on retrouve son nom dans la **correspondance de Curry-Howard**.

Regardons à nouveau les types que nous avons rencontrés et qui ne contiennent que des inconnues. Il y avait les projections `fst` et `snd` de types respectifs

$$(a \times b) \to a \qquad \text{et} \qquad (a \times b) \to b$$

Lisons $\times$ comme le *et logique* et $\to$ comme le *implique* de la logique. On obtient

a et b implique a pour le premier type, et **a et b implique b** pour le second

ou en notation logique habituelle

$$(a \wedge b) \Rightarrow a \qquad et \qquad (a \wedge b) \Rightarrow b$$

Ce sont des "vérités", des "tautologies", on parle de **formules valides** en **logique propositionnelle classique**.

Regardons maintenant le type de l'opérateur de composition avec ces nouvelles lunettes

$$(a \Rightarrow b) \Rightarrow (c \Rightarrow a) \Rightarrow c \Rightarrow b$$

C'est une formule valide! On peut écrire la table de vérité pour s'en convaincre, mais c'est un peu fastidieux… alors **prouvons** cette formule.

Supposons 1. a entraine b 2. c entraine a 3. c

Et montrons b.

De 2 et 3, je déduis a. Puis de a et 1, je déduis b. CQFD!

En fait tous les types simples du λ-calcul (le noyau d'OCaml) sont des formules valides.

> Quand on écrit une fonction OCaml d'un type ϕ, on est en fait en train d'écrire une preuve que la formule ϕ est valide!

C'est la correspondance de Curry-Howard

Monde de la logique	Monde de la programmation
Formule	Type
Preuve	Programme
Simplification de preuve	Évaluation de programme

Bon, certes, cela peut vous sembler très limité sur les types qu'on a vus. On ne sait pas comment écrire $\forall a$ dans un type, sans parler simplement du *ou logique*… on a juste entr'aperçu la base, si on veut aller plus loin il faut des systèmes de types plus compliqués.

L'un deux, la théorie des type inductifs, a donné naissance à l'assistant de preuve COQ. Une preuve en COQ contient une partie "constructive" dont on peut extraire un programme (fonctionnel, évidemment).

Pour finir: méfiez-vous quand même un peu! OCaml n'est pas aussi restrictif que le λ-calcul typé, et on peut aussi "prouver" des choses fausses en OCaml!

[79]:
```
let rec f x = f x in f
```

[79]:
```
- : 'a -> 'b = <fun>
```