

Chapitre 2 - Fonctions et récursivité

INF6120: Programmation fonctionnelle et logique

Quentin Stiévenart

Université du Québec à Montréal

v251



Type général fonction

Le type d'une fonction est dénoté par une **flèche** :

$a \rightarrow b$

représente une fonction qui

- prend une valeur de type a en argument
- renvoie une valeur de type b

Se lit « a vers b »

Définir une fonction

La construction de **base** pour définir une fonction est :

function <var> -> <expr>

Les autres variantes sont en fait du *sucre syntaxique*

Fonction anonyme

`function x -> ...` définit une **fonction anonyme** : elle n'a pas de nom

On peut associer un nom à une fonction via une liaison :

let `f = function x -> x`

Note : la fonction qui renvoie son argument est appelée la fonction *identité*

Sucre syntaxique

Pour simplifier, OCaml nous permet d'écrire :

```
let f x = x
```

Appliquer une fonction

L'appel de fonction se fait avec un espace :

```
let f = function x -> x in  
f 5
```

On a une expression à gauche et à droite de l'espace.

On aurait aussi pu écrire :

```
(function x -> x) 5
```

Appliquer une fonction : sémantique

La règle sémantique est la suivante :

- si on a une fonction $f = \text{function } x \rightarrow \text{expr}$
- et une valeur v
- $f \ v$ évalue expr dans lequel on a remplacé toutes **occurrences libres** de x par v

Appliquer une fonction : exemple

Question : quel est le résultat du programme suivant ?

```
(function x -> (x+1)) 5
```

Appliquer une fonction : exemple

Question : quel est le résultat du programme suivant ?

```
(function x -> (x+1)) 5
```

On peut noter ceci pour représenter la liaison de l'argument d'une fonction :

```
(x+1) [x -> 5] = (5+1) = 6
```

Occurrence libre

Question : quel est le résultat du programme suivant ?

```
(function x -> let x = 0 in x) 5
```

Occurrence libre

Question : quel est le résultat du programme suivant ?

```
(function x -> let x = 0 in x) 5
```

```
(let x = 0 in x) [x -> 5] = let x = 0 in x = 0
```

Car x n'est pas une occurrence libre : il est lié par le `let`

Fonction à plusieurs arguments

Comment définir une fonction qui prend plusieurs arguments ?

```
function x -> (function y -> (x + y))
```

C'est :

- une fonction qui prend un argument x
- qui renvoie une fonction qui prend un argument y
 - qui renvoie $x + y$

Appliquer une fonction à plusieurs arguments

On peut appliquer la règle sémantique sur :

```
((function x -> (function y -> (x + y))) 5) 3
```

Appliquer une fonction à plusieurs arguments

On peut appliquer la règle sémantique sur :

`((function x -> (function y -> (x + y))) 5) 3`

`((function x -> (function y -> (x + y))) 5) 3`

`= ((function y -> (5 + y))) 3`

`= (5 + 3)`

`= 8`

Type d'une fonction a plusieurs arguments

Le type de

```
((function x -> (function y -> (x + y))) 5) 3
```

est :

```
int -> (int -> int)
```

- on prend un entier en argument
- on renvoie une fonction qui prend un entier et renvoie un entier

Type d'une fonction a plusieurs arguments

Attention

`int -> (int -> int)`

est identique à :

`int -> int -> int`

mais différent de :

`(int -> int) -> int`

Fonction à plusieurs arguments ?

Il n'y a donc **que des fonctions a un seul argument**

Mais les règles de précédences nous permettent de faire *comme si* il y avait des fonctions à plusieurs arguments

```
let f = function x -> function y -> x + y in  
f 5 3
```

C'est équivalent à :

```
(f 5) 3
```

Application de fonctions

Pour **appliquer** (= appeler) une fonction, on utilise donc l'**espace** :

```
let f = function x -> function y -> x + y in  
f 5 3
```

Parenthèses et virgules

Attention aux virgules et parenthèses

```
# let f = function x -> function y -> x + y;;  
# f(5, 3);;
```

Error: This expression has **type** 'a * 'b but
an expression was expected **of type** int

On a donné un seul argument à f : un tuple de deux éléments.

Parenthèses

On peut utiliser les parenthèses pour former des sous-expressions sans ambiguïté :

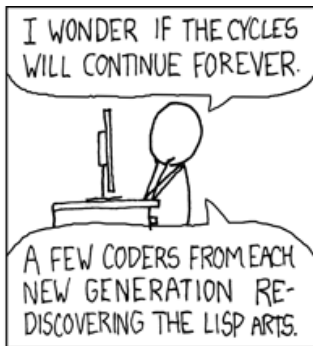
f (f 3 5) (f 8 4)

Parenthèses

Une critique souvent faite contre certains langages fonctionnels (Lisp) : trop de parenthèses

```
(defun goto-match-paren (arg)
  "Go to the matching parenthesis. Ignore ARG."
  (interactive "p")
  (cond ((looking-at "\\s\\(")
         (forward-list 1) (backward-char 1))
        ((looking-at "\\s\\)")
         (forward-char 1) (backward-list 1))
        (t (self-insert-command (or arg 1)))))
```

Parenthèses



Source: xkcd.org/297

Sucre syntaxique

Deux variantes : `fun` et `let`

`fun` permet de définir une fonction anonyme à plusieurs arguments :

```
let f = fun x y -> x + y in  
f 5 3
```

`let` permet de nommer (lier) une fonction directement :

```
let f x y = x + y in  
f 5 3
```

Mais la fonction créée est exactement la même dans tous les cas

Exercice

Réécrire l'expression suivante en utilisant uniquement la construction `function` (pas de `let` ni de `fun`)

```
let expt x y = x ** y in  
expt 2 3
```

Exercice

Réécrire l'expression suivante en utilisant uniquement la construction `function` (pas de `let` ni de `fun`)

```
let expt x y = x ** y in  
expt 2 3
```

```
(function x -> function y -> x ** y) 2 3
```

Fonction à plusieurs arguments

Une autre solution est de représenter une fonction à plusieurs arguments par une fonction qui prends un *n-uplet* en argument :

```
let f = function (x, y) -> x + y
```

Définit une fonction qui prend **1 argument** de type `int * int`

Application :

```
f (1, 2)
```

Curryification

Le fait de transformer une fonction qui prend 1 *n-uplet* en argument en une fonction qui prends *n* valeurs en argument s'appelle la **curryification**

```
let f = function (x, y) -> x + y  
let f_curried = function x -> function y -> x + y
```

La pratique en OCaml est de définir des fonctions curryifiées.

Curryification

On peut automatiser la transformation d'une variante vers l'autre en définissant deux fonctions :

val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c

val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c

Exercice : définir ces fonctions.

Curryification

On peut automatiser la transformation d'une variante vers l'autre en définissant deux fonctions :

```
val curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c
```

```
val uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c
```

Exercice : définir ces fonctions.

```
let curry (f : 'a * 'b -> 'c) : 'a -> 'b -> 'c =
```

```
  function x -> function y -> f (x, y)
```

```
let uncurry (f : 'a -> 'b -> 'c) : 'a * 'b -> 'c =
```

```
  function (x, y) -> f x y
```

Ce sont des **fonctions d'ordre supérieur** : elles prennent une fonction en argument et renvoient une nouvelle fonction.

On reviendra dessus plus tard (Chapitre 4)

Interlude: Haskell Curry



Logicien américain (1900-1982) ayant travaillé sur la logique combinatoire, en se basant sur les concepts de *Moses Schönfinkel* (1924).

Il a également travaillé sur le lien entre *programmes* et *preuves* (correspondance de *Curry-Howard*)

Occurrence libre et *shadowing*

Pour rappel :

- L'introduction d'une nouvelle variable "cache" les définitions précédentes
- Lorsqu'on évalue une variable, on prends la définition englobante la plus proche

Occurrences libres et liées

Question : dans l'expression suivantes, quelles occurrences sont libres et lesquelles sont liées ?

$x + (\text{function } y \rightarrow x + y) y$

Occurrence libre et *shadowing*

let n'est qu'un fun caché :

```
let x = 5 in x + x  
(* est équivalent à *)  
(fun x -> x+x) 5
```

Occurrence libre et *shadowing*

Exemple du labo 1 :

```
let a = 1 in  
let a = 2 and b = a in  
a + b
```

Sans let :

```
(fun a ->  
  (fun a -> (fun b -> a + b))  
  2 a) 1
```

Note : let ... and ... introduit plusieurs liaisons *en même temps*

Occurrence libre et *shadowing*

Question : quel est le résultat de l'évaluation suivante ?

```
(function x -> (function x -> x)) 1 2
```

Occurrence libre et *shadowing*

```
(function x -> (function x -> x)) 1 2
```

est équivalent à

```
let x = 1 in
```

```
let x = 2 in
```

```
x
```

Appliquer une fonction : sémantique

La règle sémantique est la suivante :

- si on a une fonction $f = \text{function } x \rightarrow \text{expr}$
- et une valeur v
- $f \ v$ évalue à expr dans lequel on a remplacé toutes **occurrences libres** de x par v

Occurrence libre et *shadowing*

```
(function x -> (function x -> x)) 1 2
```

```
->
```

```
(function x -> x) 2
```

car le `x` interne n'est pas libre, il est lié par le second `function x`

Occurrence libre et *shadowing*

À quoi évalue l'expression suivante ?

```
(function x ->  
  (function y ->  
    (function y ->  
      (x + y))  
    x)) 1 2
```

Application partielle

Soit la fonction suivante :

let add x y = x + y

Question : que vaut add 1 ?

Application partielle

Soit la fonction suivante :

```
let add x y = x + y
```

Question : que vaut `add 1` ?

```
(function x -> (function y -> x + y)) 1  
= function y -> 1 + y
```

C'est la fonction qui ajoute 1 à son argument

Opérateurs

Les opérateurs sont aussi des fonctions, juste avec un nom spécial. On peut les mettre entre parenthèses pour les traiter comme des fonctions :

```
# (+) 5 3;;
```

```
- : int = 8
```

```
# (+) 5;;
```

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

Opérateurs

Attention avec * : (* est l'ouverture de commentaire !

Ceci est invalide :

(*) 5 3

Il faut faire ceci :

(*) 5 3

Fonction et filtrage de motif

La forme `function` fait également du filtrage de motif :

function

| ... -> ...

| ... -> ...

est comme :

function x -> **match** x **with**

| ... -> ...

| ... -> ...

`match` est en fait du sucre syntaxique autour de cette forme

Fonction et filtrage de motif

```
function (x, y) -> ...
```

est comme :

```
function arg -> match arg with  
| (x, y) -> ...
```

Paramètres nommés

On peut nommer des paramètres :

```
# let f ~(x: int) : int = x*2;;  
val f : x:int -> int = <fun>  
# f ~x:5;;  
- : int = 10
```

Paramètres nommés

C'est recommandé pour éviter une ambiguïté quand plusieurs paramètres ont le même type :

```
# String.ends_with;;  
- : suffix:string -> string -> bool = <fun>  
# String.ends_with ~suffix:".ca" "uqam.ca";;  
- : bool = true  
# String.ends_with "uqam.ca" ~suffix:".ca";;  
- : bool = true
```

Paramètres avec valeurs par défauts

On peut avoir des paramètres avec des valeurs par défaut :

```
# let f ?(x: int = 5) () = x*2;;  
val f : ?x:int -> unit -> int = <fun>  
# f ();;  
- : int = 10  
# f ~x:3 ();;  
- : int = 6
```

Le () final est nécessaire si le dernier paramètre d'une fonction est avec un ?

- sinon on ne saurait pas appeler f sans le paramètre

Paramètres optionnels

On peut également avoir des paramètres optionnels, sans valeur par défaut :

```
# let f ?(x: int option) () = x;;  
val f : ?x:int -> unit -> int option = <fun>  
# f ~x:5 ();;  
- : int option = Some 5  
# f ();;  
- : int option = None
```

On verra le type option plus tard.

Fonctions : quoi utiliser en pratique ?

On va préférer :

- `let` pour nommer des fonctions
- `fun` pour quand on a une fonction anonyme
- plus rarement, `function` pour directement matcher

Fonctions et typage

OCaml possède un typage fort, avec inférence de types :

```
# let f x y = x + y;;  
val f : int -> int -> int = <fun>
```

les types de x et y sont inférés depuis le type de (+)

Dans un souci de clarté, on peut typer explicitement les paramètres et le résultat d'une fonction :

```
let f (x : int) (y : int) : int = x + y
```

Fonctions et typage

Est-il préférable de typer explicitement ou pas ?

- En pratique, on utilise des fichiers d'interface (`.mli`) qui contiennent les types
- Les fichiers `.ml` contiennent rarement beaucoup d'information de typage
- De toute façon, notre éditeur nous informe

Fonctions et typage

Dans un soucis pédagogique, on vous demande d'annoter (au moins) tout paramètre et valeur de retour des fonctions non-locales

Par exemple :

```
let sum_of_squares (x : int) (y : int) : int =  
  let square n = n * n in  
  let square_x = square x in  
  let square_y = square y in  
  square_x + square_y
```

(Les liaisons locales peuvent rester non-annotées)

Fonctions récursives

Fonctions récursives

```
let factorial (n : int) : int =  
  if n = 1 then  
    1  
  else  
    n * (factorial (n - 1))
```

Erreur lors de la définition : Unbound value factorial

factorial est une occurrence libre car par défaut let ne rend pas visible la liaison courante dans sa définition

Fonctions récursives : `let rec`

Une fonction peut être définie récursivement avec la forme de définition `let rec` :

```
let rec factorial (n : int) : int =  
  if n = 0 then  
    1  
  else  
    n * (factorial (n - 1))
```

Fonctions récursives : pow

Exercice : définir une fonction pow qui calcule la puissance y de x

```
let rec pow (x : int) (n : int) : int = ?
```

Fonctions récursives : pow

```
let rec pow (x : int) (n : int) : int =  
  if n = 0 then  
    1  
  else  
    x * (pow x (n - 1))
```

Fonctions récursives : pow

Une version plus efficace est l'*exponentiation rapide* (*exponentiation by squaring*)

Mathématiquement :

$$x^n = \begin{cases} x \times (x^2)^{(n-1)/2} & \text{si } n \text{ est impair} \\ (x^2)^{n/2} & \text{si } n \text{ est pair} \end{cases}$$

Fonctions récursives : pow

```
let rec pow (x : int) (n : int) : int =  
  match n with  
  | 0 -> 1  
  | _ when n mod 2 == 1 ->  
    x * pow (x*x) ((n-1) / 2)  
  | _ ->  
    pow (x*x) (n / 2)
```

Fonctions mutuellement récursives

Pour définir des fonctions mutuellement récursives, il faut utiliser `let rec` et `and` :

```
let rec is_even (x : int) : bool =  
  if x = 0 then  
    true  
  else  
    is_odd (x - 1)  
and is_odd (x : int) : bool =  
  not (is_even x)
```

Les liaisons introduites sont visibles dans toutes les liaisons du `let rec ... and ...`.

Récurtivité et efficacité

```
let rec factorial (n : int) : int =  
  if n = 0 then  
    1  
  else  
    n * (factorial (n - 1))
```

Quelles opérations sont calculées ? On peut représenter une *trace* des appels

Récurtivité et efficacité

```
factorial 5
= 5 * factorial 4
= 5 * 4 * factorial 3
= 5 * 4 * 3 * factorial 2
= 5 * 4 * 3 * 2 * factorial 1
= 5 * 4 * 3 * 2 * 1 * factorial 0
= 5 * 4 * 3 * 2 * 1 * 1
= 5 * 4 * 3 * 2 * 1
= 5 * 4 * 3 * 2
= 5 * 4 * 6
= 5 * 24
= 120
```

Récurtivité et efficacité

Pour calculer `factorial n`, il faut :

- `n` appels récursifs
- `n` emplacements mémoire pour stocker l'adresse de retour des appels

On peut faire mieux !

Récurtivité terminale

```
let rec factorial_iter (n : int) (acc : int) : int =  
  if n = 0 then  
    acc  
  else  
    factorial_iter (n-1) (acc * n)
```

L'appel récursif est en **position terminale** (*tail position*) : plus rien ne doit être évalué après (on ne doit donc pas stocker d'adresse de retour)

L'appel est dit récursif-terminal (*tail call*)

Récurtivité terminale

```
factorial_iter 5 1  
= factorial_iter 4 5  
= factorial_iter 3 20  
= factorial_iter 2 60  
= factorial_iter 1 120  
= factorial_iter 0 120  
= 120
```

Programmation impérative

Une fonction récursive-terminale peut facilement se compiler en code impératif tel que :

```
int factorial(int n) {  
    int acc = 1;  
    while (n > 0) {  
        acc = acc * n;  
        n = n - 1;  
    }  
    return acc;  
}
```

Réversibilité terminale et programmation impérative

La version récursive terminale et la version impérative font la même chose :

- on commence avec $acc = 1$
- on vérifie si $n = 0$
- on calcule $acc * n$
- on décrémente n
- à la fin ($n = 0$), on retourne l'accumulateur acc

Récurtivité : Fibonacci

Soit la suite de Fibonacci définie par :

$$fib(0) = 1$$

$$fib(1) = 1$$

$$fib(n) = fib(n - 1) + fib(n - 2)$$

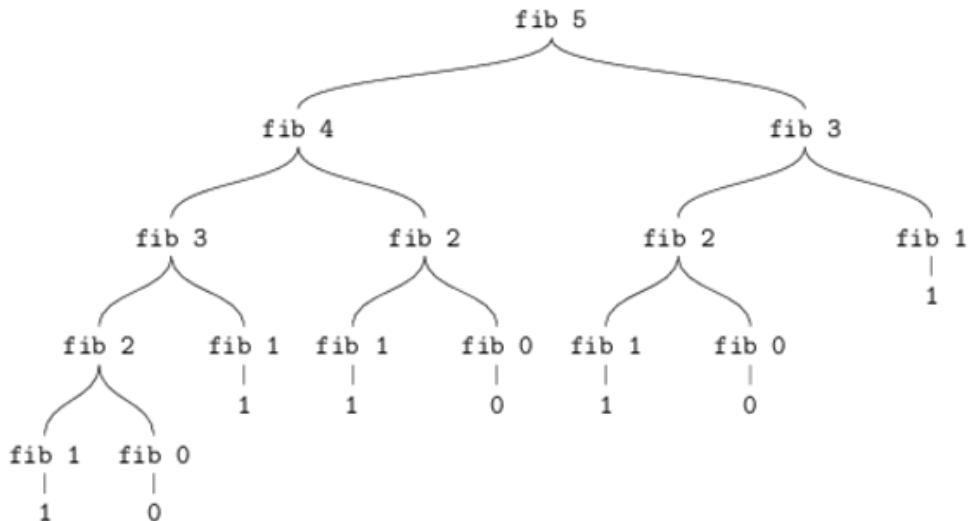
C'est la suite 0, 1, 1, 2, 3, 5, 8, ...

Récurtivité : Fibonacci

```
let rec fib (n : int) : int =  
  if n < 2 then  
    n  
  else  
    (fib (n - 1)) + (fib (n - 2))
```

Question : quel appel est en position terminale ?

Récurivité : Fibonacci



Récurtivité terminale : Fibonacci

Version récursive terminale :

```
let rec fib_iter (a : int) (b : int) (n : int) : int =  
  if n = 0 then  
    b  
  else  
    fib_iter (a + b) a (n - 1)
```

- a accumule la valeur la plus récente de la suite
- b accumule la valeur précédente de la suite

On commencera avec $a = 1$ et $b = 0$

Récurtivité terminale : Fibonacci

```
fib_iter 1 0 5 =  
fib_iter 1 1 4 =  
fib_iter 2 1 3 =  
fib_iter 3 2 2 =  
fib_iter 5 3 1 =  
fib_iter 8 5 0 =  
5
```

Récurtivité : pow

Question : est-ce que la fonction suivante est réursive-terminale ?

```
let rec pow (x : int) (y : int) : int =  
  if y = 0 then  
    1  
  else  
    x * (pow x (y - 1))
```

Fonctions locales

On peut utiliser `let ... in ...` pour introduire des fonctions locales

```
let f (x : int) : int =  
  let g y = y + 1 in  
  g (x + 1)
```

`g` est définie uniquement dans `f` : c'est une fonction **locale**

Fonctions : autres concepts

On verra plus tard (Chapitre 4) :

- les fonctions d'ordre supérieures (fonctions qui manipulent des fonctions)
- les fermetures (fonctions qui capturent des variables locales)
- les fonctions polymorphes (fonctions indépendantes du type d'argument)

Fonction mathématique

Ensemble

Un **ensemble** est une collection d'objets, ou d'éléments.

- **Opérations:** $\cup$, $\cap$, $-$, $\times$, $\mathcal{P}$, ...
- **Relations:** $\in$, $\subseteq$, ...

Opérations sur les ensembles

Si $A = \{0, 1\}$ et $B = \{1, 2, 3\}$, alors

$$A \cup B = \{0, 1, 2, 3\}$$

$$A \cap B = \{1\}$$

$$A - B = \{0\}$$

$$A \times B = \{(0, 1), (0, 2), (0, 3), (1, 1), (1, 2), (1, 3)\}$$

$$\mathcal{P}(B) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$$

Définir un ensemble par extension

$$\text{Bool} = \{\text{True}, \text{False}\}$$
$$\mathbb{N} = \{0, 1, 2, \dots\}$$
$$\text{Char} = \{\dots, '0', \dots, '9', \dots, 'A', \dots\}$$

Définir un ensemble par compréhension

$$\mathbb{Q} = \left\{ \frac{a}{b} : a \in \mathbb{Z}, b \in \mathbb{Z}^+ \right\}$$

$$[a, b] = \{x \in \mathbb{R} : a \leq x \leq b\}$$

$$\text{Char} = \{c : c \text{ est un caractère unicode}\}$$

$$\text{String} = \{s_1 s_2 \cdots s_n : n \in \mathbb{N}, s_i \in \text{Char pour } i = 1, 2, \dots, n\}$$

Définir un ensemble textuellement

Soit $A = \{0, 1\}$. Alors

- A^n est l'ensemble des chaînes de bits de longueur n
- A^* est l'ensemble des chaînes de bits (de longueurs quelconques mais finies)

Ensembles et types

- Un type est un ensemble
- $v : T \Leftrightarrow v \in T$

En Java :

- `int` est l'ensemble des entiers signés représentables sur 32 bits
- `double` est l'ensemble des flottants signés représentables sur 64 bits
- `Object` est l'ensemble de tous les objets
- `List<Integer>` est l'ensemble de toutes les listes d'entiers

Relation

Une relation (binaire) entre deux ensembles A et B est un sous-ensemble de $A \times B$

$$R \subseteq A \times B$$

Exemples :

- Relations d'équivalence, e.g., $(=) \subseteq A \times A$
- Relations d'ordre, e.g., $(\leq) \subseteq \mathbb{N} \times \mathbb{N}$

Fonction

Une **fonction** f est une relation $f \subseteq A \times B$, telle que pour tout $a \in A$, il existe **un unique** $b \in B$ tel que $(a, b) \in f$

- agit sur des valeurs venant d'un **domaine** A (un ensemble)
- donne des valeurs dans un **codomaine** B (un ensemble)

Notation :

$$f : A \rightarrow B$$
$$a \mapsto f(a)$$

Exemples de fonctions

`square` : $\mathbb{N} \rightarrow \mathbb{N}$

$x \mapsto x * x$

`(+)` : $\mathbb{R} \times \mathbb{R} \rightarrow \mathbb{R}$

$(x, y) \mapsto x + y$

`digit` : $\text{Char} \rightarrow \text{Bool}$

$c \mapsto \begin{cases} \text{True} & \text{si } c \text{ est un chiffre} \\ \text{False} & \text{sinon} \end{cases}$

Ensembles de fonctions

Les fonctions sont aussi des objets : elles peuvent être dans un ensemble.

$$\mathcal{F}_{\mathbb{N} \rightarrow \mathbb{N}} = \{f : (f : \mathbb{N} \rightarrow \mathbb{N})\}$$

$\mathcal{F}_{\text{Char} \rightarrow \text{Boolean}}$ = l'ensemble des fonctions de Char vers Boolean

Fonction récursive

Une fonction peut faire référence à elle-même.

$$\begin{aligned} fib : \mathbb{N} &\rightarrow \mathbb{N} \\ fib(n) &= \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ fib(n-1) + fib(n-2) & \text{sinon} \end{cases} \end{aligned}$$

C'est une **fonction récursive**

Fonction d'ordre supérieur

Une fonction peut prendre une autre fonction en argument :

$$\begin{aligned}\text{map} : \mathcal{F}_{\mathcal{A} \rightarrow \mathcal{B}} \times \mathcal{L}_{\mathcal{A}} &\rightarrow \mathcal{L}_{\mathcal{B}} \\ (f, L) &\mapsto [f(L[i]) : i = 0, 1, \dots, |L| - 1]\end{aligned}$$

C'est une **fonction d'ordre supérieur**

Autre exemple :

$$\begin{aligned}\text{twice} : \mathcal{F}_{\mathbb{N} \rightarrow \mathbb{N}} \times \mathbb{N} &\rightarrow \mathbb{N} \\ (f, n) &\mapsto f(f(n))\end{aligned}$$

$$\text{twice}(\text{fact}, 3) = \text{fact}(\text{fact}(3)) = \text{fact}(6) = 720$$

OCaml

On peut voir OCaml¹ comme une version exécutable des mathématiques.

```
let rec fib : int -> int = function
| 0 -> 0
| 1 -> 1
| n -> fib (n - 1) + fib (n - 2)
```

¹Son sous-ensemble pur, du moins.

λ calcul

Le **lambda calcul** est un système formel basé sur le concept de fonction.

Syntaxe

$M, N \in Term ::= x$	variable
$ \lambda x.M$	abstraction
$ M N$	application

Sémantique

$$\lambda x.M[x] \rightarrow \lambda y.M[y]$$

α -conversion

$$(\lambda x.M)N \rightarrow M[x := N]$$

β -réduction

- La première règle nous permet de renommer les variables.
- La seconde règle, d'appliquer des fonctions.

λ calcul : extensions

On peut définir un ensemble d'extensions au lambda-calcul de base :

- des types de données supplémentaires : nombres, caractères, chaînes, listes, ...
- des fonctions supplémentaires : $+$, $++$, $*$, ...
- des constructions : `if`, ...

Note: cela ne change pas l'expressivité du système

λ calcul : exemple

Notre fonction square :

$$\text{square} : \mathbb{N} \rightarrow \mathbb{N}$$

$$x \mapsto x * x$$

En λ -calcul :

$$\text{square} = \lambda x. x * x$$

λ calcul : Exemple

On peut l'appliquer :

$$\begin{aligned}\text{square } 5 &= (\lambda x. x * x) \ 5 && \text{(déf. de square)} \\ &= (x * x)[x := 5] && (\beta\text{-réduction}) \\ &= 5 * 5 && \text{(substitution)} \\ &= 25\end{aligned}$$

λ calcul : Fonction à plusieurs arguments

```
int sumOfSquares(int x, int y) {  
    return square(x) + square(y);  
}
```

s'écrit :

$$\text{sum-of-squares} = \lambda x. \lambda y. (\text{square } x) + (\text{square } y)$$

λ calcul : Fonction à plusieurs arguments

$\text{sum-of-squares} = \lambda x. \lambda y. (\text{square } x) + (\text{square } y)$

$\text{sum-of-squares } 3 \ 4$

$= (\lambda x. \lambda y. (\text{square } x) + (\text{square } y)) \ 3 \ 4$ (déf.)

$= (\lambda y. (\text{square } 3) + (\text{square } y)) [x := 3] \ 4$ (β)

$= (\lambda y. (\text{square } 3) + (\text{square } y)) \ 4$ (subst.)

$= ((\text{square } 3) + (\text{square } y)) [y := 4]$ (β)

$= ((\text{square } 3) + (\text{square } 4))$ (subst)

$= ((\lambda x. x * x) \ 3) + ((\lambda x. x * x) \ 4)$ (déf.)

$= ((x * x) [x := 3]) + ((\lambda x. x * x) \ 4)$ (β)

$= ((3 * 3) + ((\lambda x. x * x) \ 4))$ (subst.)

$= (9 + ((\lambda x. x * x) \ 4))$ (subst.)

...

$= 25$

λ calcul : Stratégie d'évaluation

$(\lambda x. \lambda y. (\lambda a. a) x) ((\lambda b. b) 1) ((\lambda c. c) 2)$

Quelle terme réduire en premier ?

C'est déterminé par la **stratégie de réduction**

λ calcul

On peut voir OCaml² comme une version exécutable du λ -calcul avec extensions et une certaine stratégie d'évaluation.

$$\lambda x. \lambda y. x + y$$

=

function x -> **function** y -> x + y

²Son sous-ensemble pur, du moins.

λ calcul

Exercice (facile)

Soit :

$$zero = \lambda f. \lambda x. x$$

$$one = \lambda f. \lambda x. f \ x$$

$$two = \lambda f. \lambda x. f \ (f \ x)$$

$$next = \lambda n. \lambda f. \lambda x. f \ ((n \ f) \ x)$$

Montrer que :

$$next \ one = two$$

λ calcul

Exercice (plus difficile)

Soit :

$$two = \lambda f. \lambda x. f (f x)$$

$$mult = \lambda m. \lambda n. \lambda f. \lambda x. (m (n f)) x$$

Et supposant qu'on ait une valeur numérique 0 et une fonction $succ = \lambda x. x + 1$, c'est-à-dire que $succ\ 0 = 1$, montrer que :

$$(((mult\ two)\ two)\ succ)\ 0 = 4$$

Concepts clés

- **Définition de fonction** : avec `function`, `fun`, `let`
- **Fonction à plusieurs arguments** : c'est juste plusieurs applications de fonctions à un argument
- **Application partielle** : quand une fonction n'est pas appliquée à tous ses arguments
- **Curryification** : transformation de fonction prenant 1 n -uplet en argument vers une fonction prenant n arguments
- **Fonction récursive** : fonction qui s'appelle elle-même, définie avec `let rec`
- **Fonction récursive terminale** : si tous les appels récurifs sont en *position terminale*. C'est mieux pour l'efficacité
- **λ calcul** : c'est la théorie fondamentale derrière le concept de fonctions des langages fonctionnels