

# Langages : interprétation et compilation

Pablo Rauzy

pr@up8.edu

[pablo.rauzy.name/teaching/liec](https://pablo.rauzy.name/teaching/liec)



UFR MITSIC / L3 informatique

Séance h

Compilation vers le  $\lambda$ -calcul

# Compilation vers le $\lambda$ -calcul

---

- ▶ On a déjà vu le  $\lambda$ -calcul en cours et en TP.
- ▶ Rappels :
  - le  $\lambda$ -calcul est un modèle de calculabilité,
  - Il comporte des variables, des fonctions anonymes (abstractions), et des applications.
  - Syntaxe :  $\Lambda ::= x, y, \dots \mid \lambda x. \Lambda \mid \Lambda \Lambda$ .

- ▶ On a déjà vu le  $\lambda$ -calcul en cours et en TP.
- ▶ Rappels :
  - le  $\lambda$ -calcul est un modèle de calculabilité,
  - Il comporte des variables, des fonctions anonymes (abstractions), et des applications.
  - Syntaxe :  $\Lambda ::= x, y, \dots \mid \lambda x. \Lambda \mid \Lambda \Lambda$ .
- ▶ Mais comment calcule-t-on effectivement avec le  $\lambda$ -calcul ?

## Réductions

- ▶ Soit  $T$  un terme de la forme  $(\lambda x. E)A$ .
- ▶  $T$  est une application dont le terme gauche est une abstraction.
- ▶ C'est ce qu'on appelle un  $\beta$ -redex.
- ▶ On peut lui appliquer une  $\beta$  contraction :
  - $T = (\lambda x. E)A$  devient alors  $T' = E[x/A]$ .
  - On note  $T \rightarrow_{\beta} T'$ .
  - $E[x/A]$  est ce qu'on appelle une  $\textit{substitution}$  :  
C'est le terme  $E$  avec les occurrences de la variable  $x$  remplacées par le terme  $A$ .

## Réductions

- ▶ Soit  $T$  un terme de la forme  $(\lambda x. E)A$ .
- ▶  $T$  est une application dont le terme gauche est une abstraction.
- ▶ C'est ce qu'on appelle un  $\beta$ -redex.
- ▶ On peut lui appliquer une  $\beta$  contraction :
  - $T = (\lambda x. E)A$  devient alors  $T' = E[x/A]$ .
  - On note  $T \rightarrow_{\beta} T'$ .
  - $E[x/A]$  est ce qu'on appelle une  $\beta$  substitution :  
C'est le terme  $E$  avec les occurrences de la variable  $x$  remplacées par le terme  $A$ .
- ▶ Si il existe une suite de termes  $T_i$  tels que  $T_0 \rightarrow_{\beta} T_1 \rightarrow_{\beta} T_2 \rightarrow_{\beta} \dots$  :
  - on dit que  $T_i$  se  $\beta$  réduit à  $T_j$  pour  $i \leq j$
  - on le note  $T_i \rightarrow_{\beta}^* T_j$ .

## Exemples

► Réduisons ensemble les termes suivants :

1.  $(\lambda x. x)y$
2.  $(\lambda x. xy)z$
3.  $(\lambda x. xx)(\lambda y. y)$
4.  $(\lambda x. xx)(\lambda x. xx)$
5.  $(\lambda x. \lambda y. x)(\lambda x. x)((\lambda x. xx)(\lambda x. xx))$
6.  $(\lambda f. \lambda x. fx)x$

## Exemples

► Réduisons ensemble les termes suivants :

1.  $(\lambda x. x)y$
2.  $(\lambda x. xy)z$
3.  $(\lambda x. xx)(\lambda y. y)$
4.  $(\lambda x. xx)(\lambda x. xx)$
5.  $(\lambda x. \lambda y. x)(\lambda x. x)((\lambda x. xx)(\lambda x. xx))$
6.  $(\lambda f. \lambda x. fx)x$

► Voyons maintenant quelques notions qui vont nous aider à formaliser tous les phénomènes ce qu'on vient de rencontrer.



## Variables libres et variables liées

- ▶ C'est la même chose en  $\lambda$ -calcul que ce qu'on a vu en programmation.
- ▶ On calcule l'ensemble des variables libres d'un terme récursivement sur sa structure :
  - $FV(x) = \{x\}$
  - $FV(\lambda x. T) = FV(T) \setminus \{x\}$
  - $FV(T_1 T_2) = FV(T_1) \cup FV(T_2)$

## Termes clos et termes ouverts

- ▶ Un terme est dit *clos* si il ne contient aucune variable libre.
- ▶ Il est ouvert sinon.
- ▶ On appelle aussi un terme clos un *combinateur*.

- ▶ Deux termes  $t$  et  $s$  sont dit  $\alpha$  *équivalents* si on peut obtenir l'un à partir de l'autre en ne faisant que renommer de variables liées.
- ▶ Il faut toutefois faire attention à ne pas *capturer* de variables libres.
- ▶ Exemples :
  - $\lambda x.x \leftrightarrow_{\alpha} \lambda y.y$
  - $\lambda x.xy \leftrightarrow_{\alpha} \lambda z.zy$
  - $\lambda x.xy \not\leftrightarrow_{\alpha} \lambda y.yy$

$\beta$ -équivalence

- ▶ On dit que  $U$  et  $V$  sont  $\beta$  *équivalents* si il existe  $T$  tels que  $U \rightarrow_{\beta}^* T$  et  $V \rightarrow_{\beta}^* T$ .
- ▶ On le note  $U =_{\beta} V$ .
- ▶ Exemples :
  - $(\lambda x. x)y =_{\beta} y$  ( $\beta$  contraction de l'un à l'autre)
  - $(\lambda x. x)y =_{\beta} (\lambda z. z)y$  (par  $\alpha$  équivalence entre les termes)
  - $(\lambda x. x)y =_{\beta} (\lambda x. y)z$  (par  $\beta$  contractions en parallèle)
  - $(\lambda f. \lambda x. fx)x =_{\beta} \lambda y. xy$  (attention à la capture de variables lors de la substitution !)

## Forme normale

- ▶ Un terme est dit en *forme normale* si il ne contient plus de  $\beta$ -rédex.
- ▶ Cette forme normale n'existe pas toujours (cf. exemple 4).
- ▶ Quand elle existe, il n'y en a qu'une seule.

# Stratégie de normalisation

- ▶ Une stratégie de normalisation correspond à une façon de choisir l'ordre dans lequel on réduit les  $\beta$ -rédux d'un terme.
- ▶ Cette stratégie a une influence sur le résultat du calcul (cf. exemple 5).
- ▶ Il existe trois stratégies principales :
  - *normal order*
  - *call-by-name*
  - *call-by-value*

## Normal order

- ▶ On commence toujours par réduire le  $\beta$ -rédux le plus externe le plus à gauche.
- ▶ Cette stratégie va toujours converger vers la forme normale.
- ▶ Aucun (?) langage de programmation n'utilise cette stratégie.

## Call-by-name

- ▶ On évalue jamais sous une abstraction.
- ▶ On évalue pas les arguments tant qu'on en a pas besoin.
- ▶ Cette stratégie ne trouve pas toujours la forme normale.
- ▶ Utilisée par quelques langages, comme  $\text{T}_{\text{E}}\text{X}$ .



# Call-by-value

- ▶ On évalue jamais sous une abstraction.
- ▶ On commence par évaluer les arguments.
- ▶ Cette stratégie trouve encore moins souvent la forme normale.
- ▶ Utilisée dans plein de langage de programmation.

Le  $\lambda$ -calcul (approche simplifiée)

- ▶ Sans prétendre avoir présenté le  $\lambda$ -calcul de manière complètement formelle et rigoureuse, on en sait maintenant assez pour jouer avec.
- ▶ Je vous ai dit plusieurs fois que c'était un langage de programmation complet.
- ▶ Prouvons-le !

- ▶ Alonzo Church est le chercheur qui a inventé le  $\lambda$ -calcul dans les années 1930.
- ▶ Il a aussi montré qu'on pouvait encoder dans ce langage toutes sortes de données et structure de données.
- ▶ Voyons quelques exemples.

- ▶ Que veut-on faire avec un booléen ?

- ▶ Que veut-on faire avec un booléen ?
  - L'utiliser pour le branchement conditionnel, c'est à dire exécuter une branche ou une autre.

- ▶ Que veut-on faire avec un booléen ?
  - L'utiliser pour le branchement conditionnel, c'est à dire exécuter une branche ou une autre.
- ▶ Du coup, on va encoder les booléens c'est des fonctions qui prennent deux arguments mais qui n'exécutent que le premier (pour `true`) ou le second (pour `false`) :
  - `true` est  $\lambda t.\lambda f.(t \_)$
  - `false` est  $\lambda t.\lambda f.(f \_)$

## Tests conditionnels

- ▶ Du coup, implémenter le `if` est trivial :

## Tests conditionnels

- ▶ Du coup, implémenter le `if` est trivial :
  - `if` est  $\lambda c.\lambda t.\lambda f.(c\ t\ f)$



## Tests conditionnels

- ▶ Du coup, implémenter le `if` est trivial :
  - `if` est  $\lambda c.\lambda t.\lambda f.(c\ t\ f)$
- ▶ On peut aussi avoir `and`, `or`, et `not` assez facilement :

- ▶ Du coup, implémenter le `if` est trivial :
  - `if` est  $\lambda c.\lambda t.\lambda f.(c\ t\ f)$
- ▶ On peut aussi avoir `and`, `or`, et `not` assez facilement :
  - `and` est  $\lambda a.\lambda b.(if\ a\ b\ false)$
  - `or` est  $\lambda a.\lambda b.(if\ a\ true\ b)$
  - `not` est  $\lambda a.(if\ a\ false\ true)$

- ▶ L'idée est de représenter chaque entier par une fonction qui prend deux arguments et applique le premier au second autant de fois que l'entier qu'on veut représenter.
  - 0 est donc  $\lambda f . \lambda z . z$ .
  - 1 est donc  $\lambda f . \lambda z . (f z)$ .
  - 2 est donc  $\lambda f . \lambda z . (f (f z))$ .
  - etc.
- ▶ Remarquez que si on passe en argument la fonction +1 et 0 (les "vraies" valeurs, pas celles encodées dans le  $\lambda$ -calcul), on obtient la ("vraie") valeur du nombre.

## Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
- succ est
  - + est
  - \* est
  - zero? est
  - pred est
  - - est

## Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
  - `+` est
  - `*` est
  - `zero?` est
  - `pred` est
  - `-` est

- On veut maintenant définir des opérations sur nos représentations des entiers :
- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
  - `+` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m f) (n f z))$
  - `*` est
  - `zero?` est
  - `pred` est
  - `-` est

## Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
  - `+` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m f) (n f z))$
  - `*` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m (n f)) z)$
  - `zero?` est
  - `pred` est
  - `-` est

## Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
  - `+` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m f) (n f z))$
  - `*` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m (n f)) z)$
  - `zero?` est  $\lambda n.(n (\lambda_.false) true)$
  - `pred` est
  - `-` est



## Opérations sur les entiers

- On veut maintenant définir des opérations sur nos représentations des entiers :
- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
  - `+` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m f) (n f z))$
  - `*` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m (n f)) z)$
  - `zero?` est  $\lambda n.(n (\lambda_.false) true)$
  - `pred` est  $\lambda n.\lambda f.\lambda z.((n (\lambda ignore1.\lambda g.(g (ignore1 f))) (\lambda_.z)) (\lambda x.x))$
  - `-` est

## Opérations sur les entiers

► On veut maintenant définir des opérations sur nos représentations des entiers :

- `succ` est  $\lambda n.\lambda f.\lambda z.(f (n f z))$
- `+` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m f) (n f z))$
- `*` est  $\lambda n.\lambda m.\lambda f.\lambda z.((m (n f)) z)$
- `zero?` est  $\lambda n.(n (\lambda_.false) true)$
- `pred` est  $\lambda n.\lambda f.\lambda z.((n (\lambda ignore1.\lambda g.(g (ignore1 f))) (\lambda_.z)) (\lambda x.x))$
- `-` est  $\lambda n.\lambda m.((m pred) n)$

► Pour les listes, on a besoin de :

- `cons` est
- `car` est
- `cdr` est
- `nil` est
- `null?` est
- `pair?` est

- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est
  - `cdr` est
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est  $\lambda p.(p\ (\lambda a.\lambda b.a))$  `error`
  - `cdr` est
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est  $\lambda p.(p\ (\lambda a.\lambda b.a))$  `error`
  - `cdr` est  $\lambda p.(p\ (\lambda a.\lambda b.b))$  `error`
  - `nil` est
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est  $\lambda p.(p\ (\lambda a.\lambda b.a))$  `error`
  - `cdr` est  $\lambda p.(p\ (\lambda a.\lambda b.b))$  `error`
  - `nil` est  $\lambda c.\lambda n.(n\ \_)$
  - `null?` est
  - `pair?` est

- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est  $\lambda p.(p\ (\lambda a.\lambda b.a))$  `error`
  - `cdr` est  $\lambda p.(p\ (\lambda a.\lambda b.b))$  `error`
  - `nil` est  $\lambda c.\lambda n.(n\ \_)$
  - `null?` est  $\lambda p.(p\ (\lambda a.\lambda b.false))\ (\lambda n.true))$
  - `pair?` est



- Pour les listes, on a besoin de :
- `cons` est  $\lambda a.\lambda b.\lambda c.\lambda n.(c\ a\ b)$
  - `car` est  $\lambda p.(p\ (\lambda a.\lambda b.a)\ \text{error})$
  - `cdr` est  $\lambda p.(p\ (\lambda a.\lambda b.b)\ \text{error})$
  - `nil` est  $\lambda c.\lambda n.(n\ \_)$
  - `null?` est  $\lambda p.(p\ (\lambda a.\lambda b.\text{false})\ (\lambda n.\text{true}))$
  - `pair?` est  $\lambda p.(p\ (\lambda a.\lambda b.\text{true})\ (\lambda n.\text{false}))$

- ▶ La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- ▶ Ce n'est pas évident, prenons un exemple.

- ▶ La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- ▶ Ce n'est pas évident, prenons un exemple.
  - `fact` est  $\lambda n.(\text{if } (\text{zero? } n) 1 (* n (\text{fact } (\text{pred } n))))$

- ▶ La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- ▶ Ce n'est pas évident, prenons un exemple.
  - `fact` est  $\lambda n.(\text{if } (\text{zero? } n) 1 (* n (\text{fact } (\text{pred } n))))$
  - Ça ne fonctionne pas, ici `fact` est une variable libre.

- ▶ La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- ▶ Ce n'est pas évident, prenons un exemple.
  - `fact` est  $\lambda n.(\text{if } (\text{zero? } n) 1 (* n (\text{fact } (\text{pred } n))))$
  - Ça ne fonctionne pas, ici `fact` est une variable libre.
  - On peut essayer de passer la fonction à elle même, `fact` serait donc :
    - $((\lambda f.\lambda n.(\text{if } (\text{zero? } n) 1 (* n ((f f) (\text{pred } n)))))$
    - $(\lambda f.\lambda n.(\text{if } (\text{zero? } n) 1 (* n ((f f) (\text{pred } n)))))$
  - Cette fois-ci ça marche !

- ▶ La seule chose qui nous reste à mettre en œuvre pour disposer d'un vrai langage de programmation est la récursion.
- ▶ Ce n'est pas évident, prenons un exemple.
  - `fact` est  $\lambda n.(\text{if } (\text{zero? } n) 1 (* n (\text{fact } (\text{pred } n))))$
  - Ça ne fonctionne pas, ici `fact` est une variable libre.
  - On peut essayer de passer la fonction à elle-même, `fact` serait donc :  
 $((\lambda f.\lambda n.(\text{if } (\text{zero? } n) 1 (* n ((f f) (\text{pred } n)))))$   
 $(\lambda f.\lambda n.(\text{if } (\text{zero? } n) 1 (* n ((f f) (\text{pred } n)))))$
  - Cette fois-ci ça marche !
  - Le combinateur `U` prend une fonction et l'applique à elle-même. On peut donc réécrire `fact` :  
 $(U (\lambda f.\lambda n.(\text{if } (\text{zero? } n) 1 (* n ((U f) (\text{pred } n)))))$

- ▶ Le combinateur U montre que c'est possible d'utiliser la récursion, mais ce qui serait vraiment bien, ce serait de pouvoir écrire facilement n'importe quelle fonction récursive.
- ▶ C'est ce que permet de faire le combinateur Y, en trouvant le point fixe d'une fonction.
  - C'est à dire que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- ▶ Il nous faut donc deux choses :
  1. exprimer une fonction récursive comme le point fixe d'une fonction pas récursive,
  2. écrire le combinateur Y.

- ▶ Si on sait qu'on reçoit la bonne fonction récursive en argument, ça devient très simple.
- ▶ En reprenant notre exemple de la fonction factorielle :
  - `fact` est  $\lambda f.\lambda n.(if (zero? n) 1 (* n (f (pred n))))$



- ▶ Dérivons le combinateur Y :
  - On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .

- Dérivons le combinateur Y :
- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
  - Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .
- Bien sûr on se retrouve avec le même problème (Y est libre !), mais aussi le fait que le calcul ne termine pas car Y se rappelle immédiatement.

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .
- Bien sûr on se retrouve avec le même problème (Y est libre !), mais aussi le fait que le calcul ne termine pas car Y se rappelle immédiatement.
- On sait que pour tout terme  $t$ ,  $t =_{\beta} \lambda x.(t x)$ .

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .
- Bien sûr on se retrouve avec le même problème (Y est libre !), mais aussi le fait que le calcul ne termine pas car Y se rappelle immédiatement.
- On sait que pour tout terme  $t$ ,  $t =_{\beta} \lambda x.(t x)$ .
- Du coup on commence par régler le second problème : Y est  $\lambda F.(F (\lambda x.((Y F) x)))$ .

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .
- Bien sûr on se retrouve avec le même problème (Y est libre !), mais aussi le fait que le calcul ne termine pas car Y se rappelle immédiatement.
- On sait que pour tout terme  $t$ ,  $t =_{\beta} \lambda x.(t x)$ .
- Du coup on commence par régler le second problème : Y est  $\lambda F.(F (\lambda x.((Y F) x)))$ .
- On sait déjà régler le premier problème avec le combinateur U : Y est  $(U (\lambda y.\lambda F.(F (\lambda x.(((U y) F) x))))))$ .

► Dérivons le combinateur Y :

- On veut que  $(Y F) =_{\beta} f$  et que  $(F f) =_{\beta} f$ .
- Par réécriture on a  $(Y F) =_{\beta} (F (Y F))$ .
- On peut utiliser une abstraction : Y est  $\lambda F.(F (Y F))$ .
- Bien sûr on se retrouve avec le même problème (Y est libre !), mais aussi le fait que le calcul ne termine pas car Y se rappelle immédiatement.
- On sait que pour tout terme  $t$ ,  $t =_{\beta} \lambda x.(t x)$ .
- Du coup on commence par régler le second problème : Y est  $\lambda F.(F (\lambda x.((Y F) x)))$ .
- On sait déjà régler le premier problème avec le combinateur U : Y est  $(U (\lambda y.\lambda F.(F (\lambda x.(((U y) F) x))))))$ .
- On peut inliner ça : Y est  $((\lambda y.\lambda F.(F (\lambda x.(y y F x)))) (\lambda y.\lambda F.(F (\lambda x.(y y F x))))))$



- ▶ On choisit un sous ensemble assez complet de Scheme :
  - les entiers, les opérations de bases sur les entiers,
  - les booléens, les opérations de bases sur les booléens,
  - les listes, les opérations de bases sur les listes,
  - le constructeur de clôture `lambda`,
  - et les constructions `let` et `letrec`.
- ▶ En utilisant ce qu'on a vu jusque là, on va le compiler vers le  $\lambda$ -calcul.
- ▶ Il nous reste quelques détails à voir.

- ▶ `(let ((var val)) body)` est équivalent à `((lambda (var) body) val)`
- ▶ On peut facilement dérouler de la même façon les assignations multiples avec `let` (en fait on implémente plutôt `let*`).

## Fonctions à plusieurs arguments

- ▶ Les fonctions de plusieurs arguments sont *currifées*.
- ▶ C'est à dire qu'on transforme une fonction de  $n$  arguments en une fonction d'un seul argument qui renvoie une clôture à  $n - 1$  argument où le nom du premier argument est lié à sa valeur.
- ▶ Exemple (en appliquant que les transformations du `let` et du `lambda`) :
  - `(lambda (a b) (+ a b))`  
devient  
`(lambda (a) (lambda (b) ((+ a) b)))`
  - donc  
`(let ((add (lambda (a b) (+ a b))))  
 (add 1 2))`  
devient  
`((lambda (add) ((add 1) 2))(lambda (a) (lambda (b) ((+ a) b)))`

- ▶ Étudions et exécutons le code ensemble.