APE 5

Exercice 1

- $\lambda x.xyz$ ok →
- $\lambda x.\lambda y$ ko → renvoie rien
- m ok
- $x\lambda wy.y$ ok
- $x\lambda$ ko
- $\lambda\lambda xz.zx$ ko
- $(mnop)(qrst)vw\lambda xyz.zxy$ ok

Exercice 2

2. **Variable libre et variable liée.** En lambda-calcul, le concept de variable libre et variable liée existe aussi. Quand une variable fait partie des arguments d'une abstraction, on dit qu'elle est liée à cette abstraction. Pour les expressions suivantes, identifiez pour chaque abstraction leur corps et indiquez pour chaque variable x à quelle abstraction est-elle liée.

- $\lambda x.\lambda y.x$
- $\lambda x.\lambda x.x$
- $\lambda x.x\lambda y.x$
- $\lambda x.x\lambda x.x$
- $\lambda z.x\lambda y.x$ tous libres
- $\lambda z.x\lambda x.x$ (↓ libre)

Exercice 3

3. **Renommage de variable** ($\alpha - conversion$)**.** Pour chaque ligne tableau suivant, indiquez si la paire d'expression est α-equivalent.

$$\lambda a.\lambda b.abb = \lambda b.\lambda a.baa$$
$$\lambda a.\lambda b.\lambda a.bb \neq \lambda i.\lambda j.jji$$
$$\lambda x.x\lambda y.x \neq \lambda e.e\lambda f.f$$
$$\lambda x.x\lambda y.x = \lambda e.e\lambda f.e$$

Exercice 4

4. **Réduction d'expression** (β)**.** Reduisez au maximum les expressions en utilisant la β-reduction

- $(\lambda x.xx)y \rightarrow (yy)$
- $(\lambda x.axxa)y \rightarrow (ayya)$
- $(\lambda x.(\lambda z.zx)q)y \rightarrow (qy)$
- $(\lambda x.x((\lambda z.zx)(\lambda x.bx)))y = yby$
- $(\lambda m.m)(\lambda n.n)(\lambda c.cc)(\lambda d.d) =$
- $\lambda z.x\lambda x.x = \lambda z.x\lambda x.x$

$(\lambda m.m)(\lambda n.n)(\lambda d.d\ \lambda d.d)$
$(\lambda m.m)(\lambda n.n)(\lambda d.d)$
$(\lambda m.m)(\lambda d.d) = \lambda d.d$

$y(\lambda z.zy)(\lambda z.bz)$
$y(\lambda z.bz)y$
yby

Exercice 5 η réduction

- $\lambda x.(\lambda y.y)x = \lambda y.y$
- $\lambda x.(\lambda y.(\lambda z.p)y)x = \lambda z.p$
- $\lambda x.(\lambda y.(\lambda z.z))x = \lambda y(\lambda z.z)$
- $\lambda x.(\lambda y.yx)p = //$
- $(\lambda f.fx)(\lambda y.gy) = (\lambda f.fx)g$

On dit d'une expression qui n'est pas réductible par β-réduction ou par η-réduction est en *forme normale* $\beta - eta$.

Exercice 6

Propriété de Church-Rosser

6. Le théorème de Church-Rosser déclare que lorsqu'on applique des règles de réduction à des termes du lambda-calcul, l'ordre dans lequel les réductions sont choisies ne fait pas de différence au résultat final.

   S'il y a deux réductions ou séquences de réductions distinctes qui peuvent être appliquées au même terme, alors il existe un terme qui peut être atteint à partir des deux résultats, en appliquant des séquences (éventuellement vides) de réductions supplémentaires.

   L'expression suivante peut être réduite de deux façons : $(\lambda x.\, \lambda y.\, x)\ ((\lambda x.\, x)\ y)$.

   - Simplifiez cette expression en réduisant d'abord les lambda-termes les plus à gauche possible. Refaites ensuite la même opération en réduisant d'abord les lambda-termes les plus à droite possible. Que remarquez-vous ?

(a) $(\lambda x.\lambda y.x)\ ((\lambda x.x)\ y) = \lambda x.x$

(b) $(\lambda x.\lambda y.x)\ ((\lambda x.x)\ y) = (\lambda x.\lambda y.x)\ y = \lambda y.y$

α équivalent

Le théorème de Church-Rosser stipule que la β-réduction est confluente. Si une expression conduit à deux formes normales irréductibles, elles sont α-équivalentes (équivalentes au renommage près).

Refaites la même procédure pour $(\lambda x.\, \lambda y.\, x\ y)\ (\lambda z.\, z)\ (\lambda w.\, w)$

(a) $(\lambda x.\lambda y.\, x\ y)\ (\lambda z.z)\ (\lambda w.w) = (\lambda z.z)(\lambda w.w) = \lambda w.w$

(b) $(\lambda x.\lambda y.\, x\ y)\ (\lambda z.\, z)\ (\lambda w.w) = (\lambda x.\lambda y.xy)\ (\lambda w.w) = \lambda y.(\lambda w.w)y$
$= \lambda w.w$

Exercice 7

7. **Arithmetiques des booléens.** Il n'y a pas de booléen ou de nombre en lambda-calcul mais il est possible de les modéliser juste avec des fonctions. Pour répresenter les booléens, on utilise une notation appelée Church-booléen:

   - $true := \lambda x.\lambda y.x$
   - $false := \lambda x.\lambda y.y$

   Avec ces lambda-termes, les opérateurs logiques peuvent être aussi définis par des lambda-termes, par exemple l'opérateur *and* peut être défini de la façon suivante:

   $$and := \lambda p.\lambda q.pqp$$

   - Définissez les opérateurs *not* et *or*.
   - Evaluez l'expression *or true false* en utilisant les lambda-termes définis au dessus.

For example: (example on the board)
AND TRUE FALSE
≡ (λp.λq.p q p) TRUE FALSE →β TRUE FALSE TRUE
≡ (λx.λy.x) FALSE TRUE →β FALSE

(a) $or := \lambda p.\lambda q.ppq \rightarrow \equiv (\lambda p.\lambda q.ppq)(TRUE\ FALSE)$
$\equiv (TRUE)\ TRUE\ FALSE \rightarrow \equiv (\lambda x\lambda y.x)\ TRUE\ FALSE \rightarrow \equiv TRUE$

(b) not := $\lambda p.p\,(FALSE)\,(TRUE) \to \equiv (\lambda p.p\,(FALSE)\,(TRUE))\,(TRUE)$

$\to \equiv (TRUE)\,(FALSE)\,(TRUE) \to \equiv (\lambda x \lambda y.x)\,(FALSE)\,(TRUE) \equiv FALSE$

Exercice 8

8. **Arithmétiques des nombres.** Pour représenter les nombres naturels, on utilise une notation appelée Church-numéral:

- $0 := \lambda f.\lambda x.x$ ou $\lambda fx.x$
- $1 := \lambda f.\lambda x.fx$ ou $\lambda fx.(fx)$
- $2 := \lambda f.\lambda x.f(fx)$ ou $\lambda fx.f(fx)$
- $3 := \lambda f.\lambda x.f(f(fx))$ ou $\lambda fx.f(f(fx)$

Un nombre n est donc représenté par une fonction à deux arguments, une fonction f et une variable x, qui execute f sur x n fois.

- Définissez une fonction *increment* qui incrémente un nombre de 1. La fonction a 3 arguments.
- La fonction *plus* est définie comme ci-dessous, en calculant *plus* 1 2, est-que le résultat est 3?
- $plus := \lambda m.\lambda n.\lambda f.\lambda x.mf(nfx)$

(a) SUCC 1 $\equiv (\lambda n \lambda f.\lambda x.f(nfx))\,(\lambda fx.(fx))$

$\equiv \lambda f.\lambda x.f(\lambda fx.(fx)\,fx)$

$\equiv \lambda f.\lambda x.f(fx) \equiv 2$

(b) PLUS 1 2 $\equiv (\lambda m.\lambda n.\lambda f.\lambda x.mf(nfx))\,(\underbrace{\lambda fx.(fx)}_{1})\,(\underbrace{\lambda fx.f(fx)}_{2})$

$\equiv (\lambda f.\lambda x.\lambda fx.(fx)\,f(\lambda fx.f(fx))\,fx)$

$\equiv \lambda f.\lambda x.\lambda fx.(fx)\,f(f(fx))$

$\equiv \lambda f.\lambda x.f(f(f(x))) \equiv 3$

Exercice 9

- $paire := \lambda a.\lambda b.\lambda f.((fa)b)$
- Définissez une fonction *first* et *second* qui prennent en argument une paire et renvoie respectivement le premier et le second élément d'une paire.
- Créer une *paire* 4 2 et appliquer vos fonction pour récupérer le premier et second élément
- En utilisant les paires, définissez une fonction *list* qui crée une liste.

(a) FIRST $\triangleq \lambda p.p\ TRUE$

SECOND $\triangleq \lambda p.p\ FALSE$

(b) PAIR 4 2 $\equiv (\lambda a.\lambda b.\lambda p.(pa\ b))\ \lambda fx.f(fx)\ \ \lambda fx.f(f(f(fx)))$

$\equiv \lambda f.((f\,\lambda fx.f(fx))\,\lambda fx\ f(f(f(fx))))$

$\equiv \lambda f.(f\ 2\ 4)$

(c) $\lambda p.(p\ 2\ 4)$ FIRST $\equiv ((\lambda p.p\ (\text{TRUE}))\ 2\ \ 4)$

$\equiv (\lambda x \lambda y.x)\ 2\ 4$

$= 2$

$\lambda p.(p\ 2\ 4)$ SECOND $\equiv ((\lambda p.p\ (\text{FALSE}))\ 2\ 4)$

$\equiv ((\lambda x.y.y)\ 2\ 4)$

$= 4$

Pairs

Let's start by focusing on pairs (or tuples). A pair is built from two arguments, a and b, and returns a function f enclosing those two arguments:

$$\text{def pair} = \lambda a.\lambda b.\lambda f.((f\ a)\ b)$$

If we bind variables a and b to values 9 and 5 we can see how a pair is constructed:

$$\begin{aligned}\text{pair } 9\ 5 &= \lambda 9.\lambda 5.\lambda f.((f\ a)\ b) \\ &= \lambda f.((f\ 9)\ 5)\end{aligned}$$

Given a pair, we can extract the first and second item using additional functions:

$$\begin{aligned}\text{def first p} &= \lambda a.\lambda b.a \\ \text{def second p} &= \lambda a.\lambda b.b\end{aligned}$$

These functions take two variables and simply return one of them. We can apply this to our pair by first creating a pair, and then applying first or second to that pair. Here we assume that p is our previously defined $\text{pair } 9\ 5$.

$$\begin{aligned}\text{p first} &= \lambda f.((f\ 9)\ 5)\ \text{first} \\ &= ((\text{first}\ 9)\ 5) \\ &= ((\lambda a.\lambda b.a)\ 9)\ 5) \\ &= \lambda 9.\lambda 5.9 \\ &= \lambda 5.9\end{aligned}$$

Lists

Now we can turn our attention to lists. A list is either empty, or consists of a head (any lambda expression) and a tail (another list). A list will consist of a pair made from the list's head and tail. That is, given the head of a list h and the tail t, the general representation of that list is simply the pair of h and t. Here, we define a list constructor in terms of pair.

$$\begin{aligned}\text{def list } h\ t &= \text{pair } h\ t \\ &= \lambda f.((f\ h)\ t)\end{aligned}$$

We can build up a bigger list by repeated applications of list. To do so, we need a termination value to represent the end of the list (nil). For now, you can use any arbitrary object to represent nil, we will assume such an object exists and call it NIL. For example, to build up a list of three elements, we invoke the list constructor three times:

$$\text{list } a\ (\text{list } b\ (\text{list } c\ \text{NIL}))$$

Exercice 10

10. **Fonction récursive**. Contrairement à un langage de programation, le lambda-calcul ne supporte pas directement la récursion en appelant une fonction f à l'intérieur de la fonction f. Le lambda-calcul utilise alors une fonction "point-fixe".

Le point fixe d'une fonction f est un élément du domaine de f, mis en correspondance avec lui-même dans f. Par exemple, "c" est le point fixe de f si $f(c) = c$. De même $f(f(...(f(c))...)) = f^n(c) = c$.

Pour la fonction $f(x) = x^2$, 0 et 1 sont les seuls points fixes de f puisque $f(0) = 0$ et $f(1) = 1$.

Nous allons utiliser ce même principe pour créer des expression lamba récursives. Il existe un combinateur qui permet de créer une version récursive d'une fonction. Ce combinateur est lui même une fonction qui retournera un point fixe pour n'importe quelle fonction que l'on lui passe :

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

Lorsque Y est utilisé comme "constructeur", elle crée un point fixe sur l'argument qui lui est passé.

Regardons ce qu'il se passe lorsque g est appliqué à Y :

$$\begin{aligned} Y\ g &= (\lambda f.(x.f\ (x\ x))\ (\lambda x.f(x\ x)))\ g \\ &= (\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x)) \\ &= g\ ((\lambda x.g\ (x\ x))\ (\lambda x.g\ (x\ x))) \\ &= g\ (Y\ g) \end{aligned}$$

Si le combinateur est appliqué indéfiniment, on obtient :

$$Y\ g = g\ (Y\ g) = g\ (g\ (Y\ g)) = g\ (...g\ (Y\ g)...)$$

Exercice: utilisez le combinateur Y pour appliquer une fonction *add* qui additionne 1 avec 1. Pour vous aider, traduisez ce pseudo-code en lambda-calcul :

```
def add f x y =
  if is_zero y then x
  else f (succ x) (pred x)
```

- Comment traduisez-vous les fonctions défninies dans la fonction `add` ? En d'autres termes, défninissez respectivement :
  - `if c then x else y`
  - `is_zero y`
  - `succ n`
  - `pred n`
- Utilisez les résultats de vos calculs précédents pour calculer en lamba calcul :
  - `is_zero` 2
  - `pred` 2
- Continuez l'évaluation de l'expression demandée :

$$Y\ add\ 1\ 1 = ...\text{(your job)}$$

Traduire l'entièreté des fonctions en lambda calcul rendrait le calcul beaucoup trop long. Utilisez les fonctions définies à l'exercice précédent pour calculer le résultat.