Disclaimer : Ce document est la résolution que j'ai faite pendant le TP, je n'ai pas vérifié l'exactitude des réponses et il est donc totalement possible que certaines résolutions soient incorrectes.

1 Définition formelle du lambda-calcul

1.1 Question 1 : Validité d'une expression.

1. $\lambda x.xyz$ est une expression valide. En effet on sait que si m est un lambda-terme et x une variable, $\lambda x.m$ est un lambda terme. En supposant ici que y et z soient des lambda termes, cette expression est valide.

2. $\lambda x.\lambda y$ est une expression valide dans laquelle l'abstraction liée à y ne contient aucunes instructions.

3. m C'est une expression lambda valide, c'est une variable.

4. $x\lambda wy.y$ est une expression valide. x est une variable (donc lambda-terme) et $\lambda wy.y$ est une abstraction, avec w comme variable libre, c'est donc également un lambda terme.

5. $x\lambda$ n'est pas une expression lambda valide.

6. $\lambda\lambda xz.zx$ n'est pas une expression lambda valide. Il faudrait écrire $\lambda xz.zx$ ou $\lambda x.\lambda z.zx$.

7. $(mnop)(qrst)vw\lambda xyz.zxy$ est une expression valide. $(mnop)$, $(qrst)$ et vw sont des lambdas termes et $\lambda xyz.zxy$ est une abstraction valide.

1.2 Question 2 : Variable libre et liée

1. $\lambda x.\lambda y.x$, la variable x est liée dans l'abstraction $\lambda x.\lambda y.x$ mais est libre dans l'abstraction $\lambda y.x$.

2. $\lambda x.\lambda x.x$, ici x est liée à l'abstraction la plus à droite.

3. $\lambda x.x\lambda y.x$, ceci peut en fait être réécrit comme : $\lambda x_1.x_1\lambda y.x_2$. On voit clairement dans cette expression que le x est lié à la première abstraction mais libre dans la seconde.

4. $\lambda x.x\lambda x.x$, le premier x est lié à la première abstraction alors que le second est lié à la seconde abstraction.

5. $\lambda z.x\lambda y.x$, ici x est libre dans les deux abstractions.

6. $\lambda z.x\lambda x.x$ ici x est lié dans la seconde abstraction définie et libre dans la première.

1.3 Question 3 : α-renaming

La première ligne est α-équivalente en effet on peut faire :

$$\lambda a.\lambda b.abb \mapsto \lambda a.\lambda c.acc \mapsto \lambda b.\lambda c.bcc \mapsto \lambda b.\lambda a.baa$$

La seconde ligne n'est pas α-équivalente, ce ne sont pas les mêmes abstractions.
La troisième ligne n'est pas α-équivalente non plus car il y'a capture de la variable x.
La dernière ligne est α-équivalente, on peut en effet faire :

$$\lambda x.x\lambda y.x \mapsto \lambda e.e\lambda y.e \mapsto \lambda e.e\lambda f.e$$

1.4 Question 4 : β-réduction

1. $(\lambda x.xx)y \mapsto yy$
2. $(\lambda x.axxa)y \mapsto ayya$
3. $(\lambda x.(\lambda z.zx)q)y \mapsto (\lambda z.zy)q \mapsto qy$
4. $(\lambda x.x((\lambda z.zx)(\lambda x.bx)))y \mapsto (\lambda x.x((\lambda x.bx)x))y \mapsto (\lambda x.x(bx))y \mapsto y(by)$
5. $(\lambda m.m)(\lambda n.n)(\lambda c.cc)(\lambda d.d) \mapsto (\lambda m.m)(\lambda n.n)((\lambda d.d)(\lambda d.d)) \mapsto \lambda d.d$

2 Propriété de Church-Rosser

2.1 Question 6

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

Simplifions d'abord cette expression en réduisant par la gauche. On a :

$$(\lambda x.\lambda y.x)((\lambda x.x)y) \mapsto_\alpha (\lambda x.\lambda a.x)((\lambda x.x)y) \mapsto_\beta \lambda a.(\lambda x.x)y \mapsto_\beta \lambda a.y$$

Simplifions maintenant par la droite :

$$(\lambda x.\lambda y.x)((\lambda x.x)y) \mapsto_\beta (\lambda x.\lambda y.x)y \mapsto_\alpha (\lambda x.\lambda a.x)y \mapsto_\beta \lambda a.y$$

3 Représentation de type de donnée

On sait définir les booléens comme des expressions lambda :

1. $true := \lambda x.\lambda y.x$, c'est une abstraction à 2 arguments qui renvoie toujours le premier.
2. $false := \lambda x.\lambda y.y$, c'est également une abstraction à 2 arguments mais cette fois-ci elle renvoie le second argument.

De ça on peut définir les opérateurs logiques, par exemple :

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

3.1 Question 7 : Arithmétique des booléens

On a que l'opérateur *not* renvoie toujours l'inverse. On peut alors le modéliser comme ceci :

$$\begin{aligned} not :=& \text{ if } b \text{ then } false \text{ else } true \\ :=& \text{ ifthenelse } b\ false\ true \\ :=& \lambda b.b\lambda yz.z\lambda yz.y \end{aligned}$$

C'est bien une focntion à un seul argument (il prend un booléen) Si ce booléen est True, renvoie la première abstraction (False) sinon renvoie la seconde (True).

Définissons maintenant l'opérateur *or*

$$\begin{aligned} or :=& \text{ if } b \text{ then } true \text{ else if } c \text{ then } true \text{ else } false \\ :=& \text{ b True c} \\ :=& \lambda bc.b\lambda yz.yc \end{aligned}$$

On a bien que cette fonction prend 2 arguments (b et c) qui sont deux booléens et rend true si au moins un des deux arguments est true et false sinon.