Lambda Calcul

III. Calcul, enfin !

In [1]:
```
from lambda_parser import parse
from lambda_terms import *
from church import *
```

Troisième notebook sur le λ-calcul. Rappelez-vous le tout début du premier notebook : j'y ai affirmé que le λ-calcul est à la base de la conception des langages de programmation fonctionnels. Nous allons voir ici, sur des exemples simples, comment la β-réduction permet de **définir** et d'**exécuter** des programmes.

1. Booléens et tests

1.1 True et False

Commençons par les deux booléens. Voici leur définition.

In [2]:
```
add_alias('TRUE', '#x.#y.x')
add_alias('FALSE', '#x.#y.y')
```

Palsambleu, pourquoi, me direz-vous ? Je répondrai "attendez". Pour que nous soyons convaincus, nous devons définir les opérations standard sur les booléens (et, ou, non) et les "instructions de test".

1.2 Opérations sur les booléens

Voici "et", "ou" et "non".

In [3]:
```
add_alias('AND', '#x.#y.xyx')
add_alias('OR', '#x.#y.xxy')
add_alias('NOT', '#b.#x.#y.byx')
```

Je vais montrer que `AND` se comporte comme il le devrait. `OR` et `NOT` sont laissés en exercice.

Théorème : Pour tout terme T,

- `AND FALSE T` $\rightarrow_\beta^*$ `FALSE`
- `AND TRUE TRUE` $\rightarrow_\beta^*$ `TRUE`

Démonstration : il suffit d'appliquer les définitions.

- `AND FALSE T` $\rightarrow_\beta^*$ `FALSE T FALSE` $\rightarrow_\beta^*$ `FALSE`
- De même, `AND TRUE TRUE` $\rightarrow_\beta^*$ `TRUE TRUE TRUE` $\rightarrow_\beta^*$ `TRUE`

Testons !

In [4]:
```
T = do('AND TRUE FALSE')
```

```
0 (λx.λy.xyx)(λx.λy.x)(λx.λy.y)
1 (λy.(λx.λy.x)y(λx.λy.x))(λx.λy.y)
2 (λx.λy.x)(λx.λy.y)(λx.λy.x)
3 (λy.λx.λy.y)(λx.λy.x)
4 λx.λy.y
--------------------
4 réductions
```

1.3 Tests

Nous allons maintenant fabriquer un terme `IF` . Nous voudrions que si B est un booléen, et T et F sont deux termes, on ait `IF B T F` $\rightarrow_\beta^*$ `T` si `B = TRUE` , et `IF B T F` $\rightarrow_\beta^*$ `F` si `B = FALSE` . Ça y est, nous avons compris le pourquoi de la définition des booléens. Ils sont faits pour rendre `IF` trivial. En fait, `B T F` sans rien devant fait le job. Histoire de faire plus rigolo, nous allons prendre pour IF ... l'identité.

In [5]:
```
add_alias('IF', '#b.#t.#f.btf')
```

In [6]:
```
T = do('IF TRUE ab')
```

```
0 (λb.λt.λf.btf)(λx.λy.x)ab
1 (λt.λf.(λx.λy.x)tf)ab
2 (λf.(λx.λy.x)af)b
3 (λx.λy.x)ab
4 (λy.a)b
5 a
--------------------
5 réductions
```

In [7]:
```
T = do('IF FALSE ab')
```

```
0 (λb.λt.λf.btf)(λx.λy.y)ab
1 (λt.λf.(λx.λy.y)tf)ab
2 (λf.(λx.λy.y)af)b
3 (λx.λy.y)ab
4 (λy.y)b
5 b
--------------------
5 réductions
```

Notre liste d'alias devient impressionnante :-).

In [8]:
```
print_aliases()
```

```
ADD λm.λn.λf.λx.mf(nfx)
SUCC λn.λf.λx.f(nfx)
PROD λm.λn.λf.λx.m(nf)x
POW λm.λn.nm
PRED λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u)
SUB λm.λn.nPRED m
TRUE λx.λy.x
FALSE λx.λy.y
AND λx.λy.xyx
OR λx.λy.xxy
NOT λb.λx.λy.byx
IF λb.λt.λf.btf
```

2. L'ordre usuel sur $\mathbb{N}$

Passons à l'ordre classique sur les entiers naturels.

2.1 Un entier est-il nul ?

Rappelez-vous la définition de l'entier de Church C_n : On a $C_n f x \rightarrow_\beta^* f^n x$. Prenons pour f la "fonction" $\lambda x. FALSE$. On a $f^0 x \rightarrow_\beta^* x$. Et si $n \geq 1, f^n x \rightarrow_\beta^* FALSE$, parce qu'on "applique" au moins une fois la fonction f. Et si on prend $x = TRUE$, on a exactement ce que l'on veut.

In [9]:
```
add_alias('ZERO', '#n.n(#x.FALSE)TRUE')
```

In [10]:
```
T = do('ZERO 3')
```

```
0 (λn.n(λx.λx.λy.y)(λx.λy.x))(λf.λx.f(f(fx)))
1 (λf.λx.f(f(fx)))(λx.λx.λy.y)(λx.λy.x)
2 (λx.(λx.λx.λy.y)((λx.λx.λy.y)((λx.λx.λy.y)x)))(λx.λy.x)
3 (λx.λx.λy.y)((λx.λx.λy.y)((λx.λx.λy.y)(λx.λy.x)))
4 λx.λy.y
--------------------
4 réductions
```

Théorème : $3 \neq 0$.

2.2 Inégalités entre entiers

La soustraction des deux entiers de Church C_m et C_n a été écrite de telle sorte que si $m \leq n$, on obtient C_0. D'où les d'inégalité LE (Lower or Equal) et GE (Greater or Equal) :

In [11]:
```
add_alias('LE', '#m.#n.ZERO (SUB m n)')
add_alias('GE', '#m.#n.ZERO (SUB n m)')
```

In [12]:
```
T = do('LE 3 2')
```

```
0 (λm.λn.(λn.n(λx.λx.λy.y)(λx.λy.x))((λm.λn.n(λn.λf.λx.n(λg.λh.h(gf)
)(λu.x)(λu.u))m)mn))(λf.λx.f(f(fx)))(λf.λx.f(fx))
1 (λn.(λn.n(λx.λx.λy.y)(λx.λy.x))((λm.λn.n(λn.λf.λx.n(λg.λh.h(gf))(λ
u.x)(λu.u))m)(λf.λx.f(f(fx)))n))(λf.λx.f(fx))
2 (λn.n(λx.λx.λy.y)(λx.λy.x))((λm.λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)
(λu.u))m)(λf.λx.f(f(fx)))(λf.λx.f(fx)))
3 (λm.λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))m)(λf.λx.f(f(fx)))(λf
.λx.f(fx))(λx.λx.λy.y)(λx.λy.x)
4 (λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(fx))))(λf.λx.
f(fx))(λx.λx.λy.y)(λx.λy.x)
5 (λf.λx.f(fx))(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(fx)))
(λx.λx.λy.y)(λx.λy.x)
6 (λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))
(λu.x)(λu.u))x))(λf.λx.f(f(fx)))(λx.λx.λy.y)(λx.λy.x)
7 (λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))((λn.λf.λx.n(λg.λh.h(gf))(λu.
x)(λu.u))(λf.λx.f(f(fx))))(λx.λx.λy.y)(λx.λy.x)
8 (λf.λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(fx)))(λg.λh
.h(gf))(λu.x)(λu.u))(λx.λx.λy.y)(λx.λy.x)
```

```
.h(gf))(λu.x)(λu.u))(λx.λx.λy.y)(λx.λy.x)
9 (λx.(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(fx)))(λg.λh.h(
g(λx.λx.λy.y)))(λu.x)(λu.u))(λx.λy.x)
10 (λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))(λf.λx.f(f(fx)))(λg.λh.h(g(λ
x.λx.λy.y)))(λu.λx.λy.x)(λu.u)
11 (λf.λx.(λf.λx.f(f(fx)))(λg.λh.h(gf))(λu.x)(λu.u))(λg.λh.h(g(λx.λx
.λy.y)))(λu.λx.λy.x)(λu.u)
12 (λx.(λf.λx.f(f(fx)))(λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.x)(λu
.u))(λu.λx.λy.x)(λu.u)
13 (λf.λx.f(f(fx)))(λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.
x)(λu.u)(λu.u)
14 (λx.(λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(λx.
λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))x)))(λu.λu.λx.λy.x)
(λu.u)(λu.u)
15 (λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(λx.λx.λ
y.y)))))((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.x)))(λu.u)
(λu.u)
16 (λh.h((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(λ
x.λx.λy.y)))))(λu.λu.λx.λy.x))(λg.λh.h(g(λx.λx.λy.y)))))(λu.u)(λu.u)
17 (λu.u)((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(
λx.λx.λy.y)))))(λu.λu.λx.λy.x))(λg.λh.h(g(λx.λx.λy.y))))(λu.u)
18 (λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))((λg.λh.h(g(λg.λh.h(g(λx.λx.λ
y.y)))))(λu.λu.λx.λy.x))(λg.λh.h(g(λx.λx.λy.y)))(λu.u)
19 (λh.h((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.x)(λg.λh.h
(g(λx.λx.λy.y)))))(λg.λh.h(g(λx.λx.λy.y)))(λu.u)
20 (λg.λh.h(g(λx.λx.λy.y)))((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.
λu.λx.λy.x)(λg.λh.h(g(λx.λx.λy.y))))(λu.u)
21 (λh.h((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.x)(λg.λh.h
(g(λx.λx.λy.y)))(λx.λx.λy.y)))(λu.u)
22 (λu.u)((λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.x)(λg.λh.
h(g(λx.λx.λy.y)))(λx.λx.λy.y))
23 (λg.λh.h(g(λg.λh.h(g(λx.λx.λy.y)))))(λu.λu.λx.λy.x)(λg.λh.h(g(λx.
λx.λy.y)))(λx.λx.λy.y)
24 (λh.h((λu.λu.λx.λy.x)(λg.λh.h(g(λx.λx.λy.y)))))(λg.λh.h(g(λx.λx.λ
y.y)))(λx.λx.λy.y)
25 (λg.λh.h(g(λx.λx.λy.y)))((λu.λu.λx.λy.x)(λg.λh.h(g(λx.λx.λy.y))))
(λx.λx.λy.y)
26 (λh.h((λu.λu.λx.λy.x)(λg.λh.h(g(λx.λx.λy.y)))(λx.λx.λy.y)))(λx.λx
.λy.y)
27 (λx.λx.λy.y)((λu.λu.λx.λy.x)(λg.λh.h(g(λx.λx.λy.y)))(λx.λx.λy.y))
28 λx.λy.y
--------------------
28 réductions
```

Et les inégalités strictes, me direz vous ? Attendez un petit peu.

2.3 Égalité de deux entiers

Enfin, un terme qui teste si deux entiers sont égaux. Il était temps ... Je rappelle que la relation $\leq$ est antisymétrique :

In [13]:
```
add_alias('EQ', '#m.#n.AND (LE m n)(LE n m)')
```

In [14]:
```
T = do('EQ 8 5', trace=False)
print_term(T)
```

```
146 réductions

λx.λy.y
```

Théorème : $8 \neq 5$:-).

2.4 Inégalités strictes

In [15]:
```
add_alias('LT','#m.#n.AND(LE m n)(NOT(EQ m n))')
add_alias('GT','#m.#n.AND(GE m n)(NOT(EQ m n))')
```

In [16]:
```
T = do('LT 3 4', trace=False)
print_term(T)
```

```
135 réductions

λx.λy.x
```

Avez-vous compris en quoi les alias nous simplifient la vie ?

3. Boucles, récursivité

3.1 La β-équivalence

Soient M et N deux termes. On dit que M et N sont β-équivalents lorsqu'il existe une suite de termes $(T_0, \dots, T_n)$ $(n \geq 0)$ telle que

- $T_0 = M, T_n = N$.
- Pour tout $i, 0 \leq i \leq n-1, T_i \rightarrow_\beta T_{i+1}$ ou $T_{i+1} \rightarrow_\beta T_i$.

C'est à dire lorsqu'on peut passer de M à N par une suite de β-réductions ou de β-réductions "inverses". Cette relation est une relation d'équivalence que nous noterons $M =_\beta N$.

3.2 Le théorème du point fixe

Théorème : Pour tout terme T il existe un terme M tel que $TM =_\beta M$.

Dit autrement, tout terme possède un point fixe !!!

Démonstration : notons $Y = UU$, où $U = \lambda u.\, \lambda x.\, x(uux)$. On a $YT = (\lambda u.\, \lambda x.\, x(uux))UT$ par définition de U. Continuons : $YT \to_\beta (\lambda u.\, \lambda x.\, x(uux))[u := U]T = (\lambda x.\, x(UUx))T$. Encore une petite β-réduction :
$(\lambda x.\, x(UUx))T \to_\beta (x(UUx))[x := T] = T(UUT) = T(YT)$.

Ainsi, en posant $M = YT$, on a $M \to_\beta^* TM$, et donc $TM =_\beta M$. Non seulement on a prouvé l'existence d'un point fixe, mais on en possède un explicitement.

Le terme `Y` est appelé le combinateur de point fixe de Turing. Il est essentiel pour ce que nous allons faire dans la suite.

In [17]:
```
add_alias('Y', '(#u.#x.x(uux))(#u.#x.x(uux))')
```

3.3 Boucles et récursivité

Considérons la fonction f qui, à tout $n \in \mathbb{N}$, associe $\sum_{k=0}^{n} k$. Et supposons que l'on ne sait pas combien fait la somme parce qu'on ne connaît pas son cours de maths. La définition en Python de f est immédiate :

In [18]:
```
def f(n):
    s = 0
    for k in range(n + 1): s = s + k
    return s
```

In [19]:
```
[f(n) for n in range(10)]
```

Out[19]:
```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

Imaginons maintenant que Python ne connaisse ni boucles `for` ni boucles `while`. Que faire ? Il suffit d'utiliser la récursivité ! La plupart des langages fonctionnels n'ont PAS d'instructions de boucle ... et ils peuvent faire autant de choses que Python.

De façon un peu lapidaire : **boucles ↔ tests + récursivité**

In [20]:
```
def f1(n):
    if n == 0: return 0
    else: return n + f1(n - 1)
```

In [21]:
```
[f1(n) for n in range(10)]
```

Out[21]:
```
[0, 1, 3, 6, 10, 15, 21, 28, 36, 45]
```

Revenons au λ-calcul. Les tests, on sait faire. Mais un terme ne peut pas être récursif ? Non, effectivement, nous devons réfléchir de façon un peu plus profonde à ce qu'est vraiment la récursivité.

Une fonction récursive, d'une variable x, mettons, est une fonction f définie par $f = x \mapsto \Phi(f, x)$ où Φ est une expression plus ou moins compliquée. En termes de termes (je l'ai déjà faite celle-là, désolé), $f = \lambda x.\, \Phi f x$. Considérons le terme $\Psi = \lambda f.\, \lambda x.\, \Phi f x$. On a donc $f = \Psi f$, c'est à dire que ... f est un point fixe de Ψ !. Sans entrer dans des considérations, pourtant essentielles, d'unicité, nous en déduisons que $f = Y\Psi$, où Y est le combinateur de Turing. Je me résume.

- Une fonction récursive est la solution d'une équation $f = \Psi f$.
- Une solution de cette équation est $Y\Psi$.
- Si cette équation (sous certaines hypothèses), a une unique solution, alors $f = Y\Psi$.

Bilan : Pour coder f il faut trouver le terme Ψ.

Faisons le pour $f(n) =$ la somme des entiers de 0 à n. Les égalités qui suivent sont en fait des β-équivalences. On a

" `f(n) = SI(n = 0) ALORS(0) SINON(n + f(n - 1))` ". Sans guillemets (ou presque) :

`f n = IF(ZERO n)0(ADD n (f (SUB n 1))` . On brûle :

`f = #n.IF(ZERO n)0(ADD n (f (SUB n 1))` . Encore un effort :

`f = (#f.#n.IF(ZERO n)0(ADD n (f (SUB n 1)))f` . On y est !

`f = Tf` où

`T = #f.#n.IF(ZERO n)0(ADD n (f (SUB n 1))` . Ainsi,

`f = YT` .

In [22]:
```
add_alias('FSUM', 'Y#f.#n.IF(ZERO n)0(ADD n(f(PRED n)))')
```

In [23]:
```
print_term(do('FSUM 10', max_red=10000, trace=False))
```

```
1847 réductions

λf.λx.f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(fx)))))))))))))))))))))
))))))))))))))))))))))))))))))))))
```

3.4 Les nombres de Fibonacci

Je laisse le lecteur se convaincre que le terme ci-dessous calcule les nombres de Fibonacci, définis par

- $F_0 = 0, F_1 = 1$
- $\forall n \geq 2, F_n = F_{n-1} + F_{n-2}$

In [24]:
```
add_alias('FIB', 'Y#f.#n.IF(LE n 1)n((ADD (f (PRED n))(f (PRED (PRED n
```

In [25]:
```
T = do('FIB 9', max_red=100000, trace=False)
print_term(T)
```

```
21972 réductions

λf.λx.f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(f(
f(f(fx))))))))))))))))))))))))))))))))))
```

In [26]:
```
unchurch(T)
```

Out[26]:
```
34
```

Je vous déconseille d'essayer de calculer F_{50} :-).

4. Couples, triplets, listes

4.1 Couples

Comment modéliser les couples ? Voici la solution.

In [27]:
```
add_alias('PAIR', '#x.#y.#z.zxy')
add_alias('FST', '#p.p TRUE')
add_alias('SND', '#p.p FALSE')
```

In [28]:
```
do('PAIR a b')
```

```
0 (λx.λy.λz.zxy)ab
1 (λy.λz.zay)b
2 λz.zab
--------------------
2 réductions
```

Out[28]:
```
[2, 'z', [1, [1, [0, 'z'], [0, 'a']], [0, 'b']]]
```

In [29]:
```
do('FST(PAIR a b)')
```

```
0 (λp.p(λx.λy.x))((λx.λy.λz.zxy)ab)
1 (λx.λy.λz.zxy)ab(λx.λy.x)
2 (λy.λz.zay)b(λx.λy.x)
3 (λz.zab)(λx.λy.x)
4 (λx.λy.x)ab
5 (λy.a)b
6 a
--------------------
6 réductions
```

Out[29]:
```
[0, 'a']
```

In [30]:
```
do('SND(PAIR a b)')
```

```
0 (λp.p(λx.λy.y))((λx.λy.λz.zxy)ab)
1 (λx.λy.λz.zxy)ab(λx.λy.y)
2 (λy.λz.zay)b(λx.λy.y)
3 (λz.zab)(λx.λy.y)
4 (λx.λy.y)ab
5 (λy.y)b
6 b
--------------------
6 réductions
```

Out[30]:
```
[0, 'b']
```

4.2 Triplets

Et les triplets ? On peut représenter un triplet (a, b, c) par le couple $(a, (b, c))$.

In [31]:
```
do('FST(PAIR a (PAIR b c))')
```

```
0 (λp.p(λx.λy.x))((λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc))
1 (λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc)(λx.λy.x)
2 (λy.λz.zay)((λx.λy.λz.zxy)bc)(λx.λy.x)
3 (λz.za((λx.λy.λz.zxy)bc))(λx.λy.x)
4 (λx.λy.x)a((λx.λy.λz.zxy)bc)
5 (λy.a)((λx.λy.λz.zxy)bc)
6 a
--------------------
6 réductions
```

Out[31]:
```
[0, 'a']
```

In [32]:
```
do('FST(SND(PAIR a (PAIR b c)))')
```

```
0 (λp.p(λx.λy.x))((λp.p(λx.λy.y))((λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc))
)
1 (λp.p(λx.λy.y))((λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc))(λx.λy.x)
2 (λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc)(λx.λy.y)(λx.λy.x)
3 (λy.λz.zay)((λx.λy.λz.zxy)bc)(λx.λy.y)(λx.λy.x)
4 (λz.za((λx.λy.λz.zxy)bc))(λx.λy.y)(λx.λy.x)
5 (λx.λy.y)a((λx.λy.λz.zxy)bc)(λx.λy.x)
6 (λy.y)((λx.λy.λz.zxy)bc)(λx.λy.x)
7 (λx.λy.λz.zxy)bc(λx.λy.x)
8 (λy.λz.zby)c(λx.λy.x)
9 (λz.zbc)(λx.λy.x)
10 (λx.λy.x)bc
11 (λy.b)c
12 b
--------------------
12 réductions
```

Out[32]:
```
[0, 'b']
```

In [33]:
```
do('SND(SND(PAIR a (PAIR b c)))')
```

```
0 (λp.p(λx.λy.y))((λp.p(λx.λy.y))((λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc))
)
1 (λp.p(λx.λy.y))((λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc))(λx.λy.y)
2 (λx.λy.λz.zxy)a((λx.λy.λz.zxy)bc)(λx.λy.y)(λx.λy.y)
3 (λy.λz.zay)((λx.λy.λz.zxy)bc)(λx.λy.y)(λx.λy.y)
4 (λz.za((λx.λy.λz.zxy)bc))(λx.λy.y)(λx.λy.y)
5 (λx.λy.y)a((λx.λy.λz.zxy)bc)(λx.λy.y)
6 (λy.y)((λx.λy.λz.zxy)bc)(λx.λy.y)
7 (λx.λy.λz.zxy)bc(λx.λy.y)
8 (λy.λz.zby)c(λx.λy.y)
9 (λz.zbc)(λx.λy.y)
10 (λx.λy.y)bc
11 (λy.y)c
12 c
--------------------
12 réductions
```

Out[33]:
```
[0, 'c']
```

5.3 La division dans $\mathbb{N}$

Soient $a, b \in \mathbb{N}, b \neq 0$. Si $a < b$, le quotient de la division de a par b est 0. Sinon, soit q' le quotient de la division de $a - b$ par b. Le quotient de la division de a par b est alors $q' + 1$.

Ainsi, une division se fait par soustractions successives. Hautement inefficace, mais faisable !

In [34]:
```
add_alias('DIV', 'Y#f.#a.#b.IF(LT a b)0(ADD (f (SUB a b) b) 1)')
```

In [35]:
```
print_term(do('DIV 11 3', trace=False, max_red=10000))
```

```
1314 réductions

λf.λx.f(f(fx))
```

Voici de même un terme qui calcule le reste de la division.

In [36]:
```
add_alias('MOD', 'Y#f.#a.#b.IF(LT a b)a(f (SUB a b) b)')
```

In [37]:
```
print_term(do('MOD 11 3', trace=False, max_red=10000))
```

```
1479 réductions

λf.λx.f(fx)
```

Et un terme qui renvoie la paire (quotient, reste) ...

In [38]:
```
add_alias('DIVMOD', '#a.#b.PAIR(DIV a b)(MOD a b)')
```

In [39]:
```
print_term(do('DIVMOD 11 3', max_red=10000, trace=False))
```

```
2797 réductions

λz.z(λf.λx.f(f(fx)))(λf.λx.f(fx))
```

Exercice : écrire un terme qui calcule le PGCD de deux entiers.

5.4 Listes

5.4.1 Fonctions de base

Tout d'abord, qu'est-ce qu'une liste ? Je ne parle pas des listes en Python, mais de la structure de liste utilisée par les langages fonctionnels comme Haskell ou Caml. Le type "liste" est défini récursivement.

Il existe une liste "vide", que nous noterons NIL. Et toute liste non vide possède une **tête**, qui est un objet, et une **queue**, qui est une liste. Si nous voulons manipuler des listes nous avons besoin des fonctions suivantes :

- $CONS$, qui prend un objet a et une liste s, et qui renvoie la liste dont la tête est a et la queue est s.
- HD, qui prend une liste et renvoie sa tête.
- TL, qui prend une liste et renvoie sa queue.
- $ISNIL$ qui prend une liste et nous dit si elle est vide.

Quel λ-terme pourrait modéliser la liste $[a_0, \dots, a_n]$? Rappelons-nous les entiers de Church : on "stockait" d'une certaine façon l'entier n dans le terme $f^n x$, où f et x sont des variables. Ici, l'idée est de stocker les a_i dans le terme $fa_1(fa_2(\dots(fa_{n-1}x)))$. Et pour NIL ? Prenons C_0, tout simplement.

In [40]:
```
add_alias('NIL', '#f.#x.x')
add_alias('CONS', '#a.#s.#f.#x.fa(sfx)')
```

Voici par exemple comment créer la liste $[a, b, c, d]$:

In [41]:
```
do('CONS a(CONS b (CONS c (CONS d NIL)))')
```

```
0 (λa.λs.λf.λx.fa(sfx))a((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx)
)c((λa.λs.λf.λx.fa(sfx))d(λf.λx.x))))
1 (λs.λf.λx.fa(sfx))((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((
λa.λs.λf.λx.fa(sfx))d(λf.λx.x))))
2 λf.λx.fa((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λ
x.fa(sfx))d(λf.λx.x)))fx)
3 λf.λx.fa((λs.λf.λx.fb(sfx))((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.fa
(sfx))d(λf.λx.x)))fx)
4 λf.λx.fa((λf.λx.fb((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.fa(sfx))d(λ
f.λx.x))fx))fx)
5 λf.λx.fa((λx.fb((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.fa(sfx))d(λf.λ
x.x))fx))x)
6 λf.λx.fa(fb((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.fa(sfx))d(λf.λx.x)
)fx))
7 λf.λx.fa(fb((λs.λf.λx.fc(sfx))((λa.λs.λf.λx.fa(sfx))d(λf.λx.x))fx)
)
8 λf.λx.fa(fb((λf.λx.fc((λa.λs.λf.λx.fa(sfx))d(λf.λx.x)fx))fx))
9 λf.λx.fa(fb((λx.fc((λa.λs.λf.λx.fa(sfx))d(λf.λx.x)fx))x))
10 λf.λx.fa(fb(fc((λa.λs.λf.λx.fa(sfx))d(λf.λx.x)fx)))
11 λf.λx.fa(fb(fc((λs.λf.λx.fd(sfx))(λf.λx.x)fx)))
12 λf.λx.fa(fb(fc((λf.λx.fd((λf.λx.x)fx))fx)))
13 λf.λx.fa(fb(fc((λx.fd((λf.λx.x)fx))x)))
14 λf.λx.fa(fb(fc(fd((λf.λx.x)fx))))
15 λf.λx.fa(fb(fc(fd((λx.x)x))))
16 λf.λx.fa(fb(fc(fdx)))
--------------------
16 réductions
```

Out[41]:
```
[2,
 'f',
 [2,
  'x',
  [1,
   [1, [0, 'f'], [0, 'a']],
   [1,
    [1, [0, 'f'], [0, 'b']],
    [1, [1, [0, 'f'], [0, 'c']], [1, [1, [0, 'f'], [0, 'd']], [0, 'x
']]]]]]]
```

Récupérer la tête d'une liste, c'est facile. Si s est une liste non vide, et qu'on l'applique à la fonction f constante égale à TRUE et à x quelconque, on obtient a_0 . Remarquons que $HD\ NIL$ nous donnera x, mais la tête de la liste vide n'est pas censée exister.

In [42]:
```
add_alias('HD', '#s.s TRUE 0')
```

In [43]:
```
do('HD NIL')
```

```
0 (λs.s(λx.λy.x)(λf.λx.x))(λf.λx.x)
1 (λf.λx.x)(λx.λy.x)(λf.λx.x)
2 (λx.x)(λf.λx.x)
3 λf.λx.x
--------------------
3 réductions
```

Out[43]:
```
[2, 'f', [2, 'x', [0, 'x']]]
```

In [44]:
```
do('HD(CONS a(CONS b (CONS c (CONS d NIL))))')
```

```
0 (λs.s(λx.λy.x)(λf.λx.x))((λa.λs.λf.λx.fa(sfx))a((λa.λs.λf.λx.fa(sf
x))b((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.fa(sfx))d(λf.λx.x)))))
1 (λa.λs.λf.λx.fa(sfx))a((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx)
)c((λa.λs.λf.λx.fa(sfx))d(λf.λx.x))))(λx.λy.x)(λf.λx.x)
2 (λs.λf.λx.fa(sfx))((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((
λa.λs.λf.λx.fa(sfx))d(λf.λx.x))))(λx.λy.x)(λf.λx.x)
3 (λf.λx.fa((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.
λx.fa(sfx))d(λf.λx.x)))fx))(λx.λy.x)(λf.λx.x)
4 (λx.(λx.λy.x)a((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((λa.λ
s.λf.λx.fa(sfx))d(λf.λx.x)))(λx.λy.x)x))(λf.λx.x)
5 (λx.λy.x)a((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf
.λx.fa(sfx))d(λf.λx.x)))(λx.λy.x)(λf.λx.x))
6 (λy.a)((λa.λs.λf.λx.fa(sfx))b((λa.λs.λf.λx.fa(sfx))c((λa.λs.λf.λx.
fa(sfx))d(λf.λx.x)))(λx.λy.x)(λf.λx.x))
7 a
--------------------
7 réductions
```

Out[44]:
```
[0, 'a']
```

Comment tester si une liste est vide ? Eh bien $ISNIL$ c'est comme $CONS$ sauf que pour f on prend la fonction constante égale à FALSE, et pour x on prend TRUE. Du coup, le petit inconvénient que nous avons remarqué tout à l'heure concernant $HD\ NIL$ devient un gros avantage.

In [45]:
```
add_alias('ISNIL', '#s.s(#a.#b.FALSE)TRUE')
```

In [46]:
```
do('ISNIL (CONS a NIL)')
```

```
0 (λs.s(λa.λb.λx.λy.y)(λx.λy.x))((λa.λs.λf.λx.fa(sfx))a(λf.λx.x))
1 (λa.λs.λf.λx.fa(sfx))a(λf.λx.x)(λa.λb.λx.λy.y)(λx.λy.x)
2 (λs.λf.λx.fa(sfx))(λf.λx.x)(λa.λb.λx.λy.y)(λx.λy.x)
3 (λf.λx.fa((λf.λx.x)fx))(λa.λb.λx.λy.y)(λx.λy.x)
4 (λx.(λa.λb.λx.λy.y)a((λf.λx.x)(λa.λb.λx.λy.y)x))(λx.λy.x)
5 (λa.λb.λx.λy.y)a((λf.λx.x)(λa.λb.λx.λy.y)(λx.λy.x))
6 (λb.λx.λy.y)((λf.λx.x)(λa.λb.λx.λy.y)(λx.λy.x))
7 λx.λy.y
--------------------
7 réductions
```

Out[46]:
```
[2, 'x', [2, 'y', [0, 'y']]]
```

In [47]:
```
do('ISNIL NIL')
```

```
0 (λs.s(λa.λb.λx.λy.y)(λx.λy.x))(λf.λx.x)
1 (λf.λx.x)(λa.λb.λx.λy.y)(λx.λy.x)
2 (λx.x)(λx.λy.x)
3 λx.λy.x
--------------------
3 réductions
```

Out[47]:
```
[2, 'x', [2, 'y', [0, 'x']]]
```

Reste à coder TL. Et là on tremble parce qu'on se souvient de $PRED$ pour les entiers de Church. Effectivement, c'est tout aussi problématique, et je ne justifierai pas le choix du terme que nous allons choisir.

In [48]:
```
add_alias('TL', '#s.FST(s(#a.#b.PAIR(SND b)(CONS a (SND b)))(PAIR NIL
```

In [49]:
```
T = do('TL(CONS a(CONS b (CONS c (CONS d NIL))))', trace=False)
print_term(T)
```

```
69 réductions

λf.λx.fb(fc(fdx))
```

5.4.2 Longueur d'une liste

Ouf, nous y sommes. Peut-on maintenant écrire des fonctions qui opèrent sur les listes ? Tentons la fonction qui prend une liste et renvoie sa longueur. La liste vide est de longueur 0, et la longueur d'une liste non vide est $\ell(t) + 1$ où t est la queue de la liste. Un petit coup de combinateur de Turing et nous y sommes :

In [50]:
```
add_alias('LEN', 'Y#f.#s.IF(ISNIL s)0(ADD (f(TL s))1)')
```

In [51]:
```
T = do('LEN (CONS a (CONS b (CONS c NIL)))', trace=False)
print_term(T)
```

```
264 réductions

λf.λx.f(f(fx))
```

Super, la liste $[a, b, c]$ est de longueur 3 !

5.4.3 Concaténation de listes

Encore un petit effort avant le Graal : une fonction qui concatène des listes. Si nous l'appelons provisoirement γ, nous avons, pour toutes listes s, s' :

- $\gamma[]s' = s'$
- $\gamma ss' = CONS\ (HD\ s)\gamma(TL\ s)s'$ si s n'est pas vide.

Le terme `CONCAT` fait le boulot :

In [52]:
```
add_alias('CONCAT', 'Y(#f.#s.#t.IF(ISNIL s)t(CONS (HD s)(f (TL s)t)))'
```

In [53]:
```
T = do('CONCAT (CONS a (CONS b NIL))(CONS c (CONS d (CONS e NIL)))', t
print_term(T)
```

```
185 réductions

λf.λx.fa(fb(fc(fd(fex))))
```

Exercice : écrire un terme qui renverse les listes.

5.5 Les tours de Hanoi

Nous allons finir ce notebook en beauté en résolvant le problème des tours de Hanoi.

Nous avons trois piquets et n disques de tailles différentes. Les disques sont empilés dans l'ordre de taille décroissante sur un des trois piquets. Le but du jeu est de déplacer tous les disques sur un autre piquet choisi à l'avance, en appliquant les règles ci-dessous :

- On ne peut déplacer qu'un disque à la fois, d'un piquet vers un autre piquet.
- Un disque ne peut pas être posé sur un disque de taille plus petite.

Ce jeu a une solution récursive évidente. Pour déplacer n disques du piquet a vers le piquet b :

- S'il n'y a qu'un disque c'est facile.
- Sinon :
  - soit c le troisième piquet.
  - déplacer $n-1$ disques du piquet a vers le piquet c.
  - déplacer le dernier disque du piquet a vers le piquet b.
  - déplacer $n-1$ disques du piquet c vers le piquet b.

En Python cela donne le code suivant. Il suffit de remarquer que si les piquets sont numérotés 0,1,2, la somme des numéros vaut 3. Donc, si on m'en donne deux, a et b, il est facile de trouver le troisième. C'est $c = 3 - a - b$. La fonction `hanoi` prend en paramètres le nombre de disques, n, et les numéros des piquets de départ et d'arrivée, a et b. Elle renvoie la liste des mouvements à faire pour résoudre le problème.

In [54]:
```
def hanoi(n, a, b):
    if n == 1: return [(a,b)]
    else:
        c = 3 - a - b
        return hanoi(n-1, a, c) + hanoi(1, a, b) + hanoi(n-1, c, b)
```

In [55]:
```
print(hanoi(4,0,1))
```

```
[(0, 2), (0, 1), (2, 1), (0, 2), (1, 0), (1, 2), (0, 2), (0, 1), (2,
1), (2, 0), (1, 0), (2, 1), (0, 2), (0, 1), (2, 1)]
```

Et voici le λ-terme qui fait le travail. Je vous laisse le lire et le comprendre.

In [56]:
```
add_alias('MOVE', '#a.#b.CONS(PAIR a b)NIL')
add_alias('HAN', '#f.#n.#a.#b.IF(EQ n 1)(MOVE a b)(CONCAT (f (PRED n)a
add_alias('HANOI', 'Y HAN')
```

Voici le programme Hanoi dans toute sa lambdatitude :

In [57]:
```
print_term(expand(parse('HANOI')))
```

```
(λu.λx.x(uux))(λu.λx.x(uux))(λf.λn.λa.λb.(λb.λt.λf.btf)((λm.λn.(λx.λ
y.xyx)((λm.λn.(λn.n(λx.λx.λy.y)(λx.λy.x))((λm.λn.n(λn.λf.λx.n(λg.λh.
h(gf))(λu.x)(λu.u))m)mn))mn)((λm.λn.(λn.n(λx.λx.λy.y)(λx.λy.x))((λm.
λn.n(λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))m)mn))nm))n(λf.λx.fx))((λa.
λb.(λa.λs.λf.λx.fa(sfx))((λx.λy.λz.zxy)ab)(λf.λx.x))ab)((λu.λx.x(uux
))(λu.λx.x(uux))(λf.λs.λt.(λb.λt.λf.btf)((λs.s(λa.λb.λx.λy.y)(λx.λy.
x))s)t((λa.λs.λf.λx.fa(sfx))((λs.s(λx.λy.x)(λf.λx.x))s)(f((λs.(λp.p(
λx.λy.x))(s(λa.λb.(λx.λy.λz.zxy)((λp.p(λx.λy.y))b)((λa.λs.λf.λx.fa(s
fx))a((λp.p(λx.λy.y))b)))((λx.λy.λz.zxy)(λf.λx.x)(λf.λx.x))))s)t))))(
f((λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))n)a((λm.λn.n(λn.λf.λx.n(λg.λh
.h(gf))(λu.x)(λu.u))m)(λf.λx.f(f(fx)))((λm.λn.λf.λx.mf(nfx))ab)))((λ
a.λs.λf.λx.fa(sfx))((λa.λb.(λa.λs.λf.λx.fa(sfx))((λx.λy.λz.zxy)ab)(λ
f.λx.x))ab)(f((λn.λf.λx.n(λg.λh.h(gf))(λu.x)(λu.u))n)((λm.λn.n(λn.λf
.λx.n(λg.λh.h(gf))(λu.x)(λu.u))m)(λf.λx.f(f(fx)))((λm.λn.λf.λx.mf(nf
x))ab))b))))
```

In [58]:
```
T = do('HANOI 3 0 1', trace=False, max_red=100000)
print_term(T)
```

```
3531 réductions

λf.λx.f(λz.z(λf.λx.x)(λf.λx.fx))(f(λf.λx.f(λz.z(λf.λx.x)(λf.λx.f(fx)
))x)(f(λz.z(λf.λx.fx)(λf.λx.f(fx)))(f(λf.λx.f(λz.z(λf.λx.x)(λf.λx.fx
))x)(f(λz.z(λf.λx.f(fx))(λf.λx.x))(f(λf.λx.f(λz.z(λf.λx.f(fx))(λf.λx
.fx))x)(f(λz.z(λf.λx.x)(λf.λx.fx))x))))))
```

Combien de déplacements de disques ?

In [59]:
```
T = do('LEN(HANOI 3 0 1)', trace=False, max_red=100000)
print_term(T)
```

```
12636 réductions

λf.λx.f(f(f(f(f(f(fx))))))
```

In [60]:
```
unchurch(T)
```

Out[60]: 7

Tentons 4 disques. Mais ce sera bien le maximum que l'on puisse faire. Soyez patients.

In [61]:
```
T = do('HANOI 4 1 2', trace=False, max_red=1000000)
print_term(T)
```

```
34551 réductions

λf.λx.f(λz.z(λf.λx.fx)(λf.λx.x))(f(λf.λx.f(λz.z(λf.λx.fx)(λf.λx.f(fx
)))x)(f(λz.z(λf.λx.x)(λf.λx.f(fx)))(f(λf.λx.f(λz.z(λf.λx.fx)(λf.λx.x
))x)(f(λz.z(λf.λx.f(fx))(λf.λx.fx))(f(λf.λx.f(λz.z(λf.λx.f(fx))(λf.λ
x.x))x)(f(λz.z(λf.λx.fx)(λf.λx.x))(f(λf.λx.f(λz.z(λf.λx.fx)(λf.λx.f(
fx)))x)(f(λz.z(λf.λx.x)(λf.λx.f(fx)))(f(λf.λx.f(λz.z(λf.λx.x)(λf.λx.
fx))x)(f(λz.z(λf.λx.f(fx))(λf.λx.fx))(f(λf.λx.f(λz.z(λf.λx.x)(λf.λx.
f(fx)))x)(f(λz.z(λf.λx.fx)(λf.λx.x))(f(λf.λx.f(λz.z(λf.λx.fx)(λf.λx.
f(fx)))x)(f(λz.z(λf.λx.x)(λf.λx.f(fx)))x))))))))))))))
```

Voilà la solution. Ce n'est pas lisible mais on a PU le faire.

6. Le mot de la fin

Vous avez maintenant une idée de ce qu'est le λ-calcul. Si l'on veut rester dans les applications du λ-calcul à l'informatique, l'étape suivante serait sans aucun doute d'aborder le λ-calcul typé. Dans les langages fonctionnels comme Caml ou Haskell, les objets ont un "type" et ce type peut être **inféré** par le compilateur. Par exemple, en Caml, la fonction `fun g f x -> g (f x)` est de type $(b \to c) \to (a \to b) \to a \to c$.

La question de l'inférence de type est passionnante, elle est liée à un algorithme très général qui s'appelle l'algorithme d'unification. La forme des types possibles pour les objets conduit à des théorèmes profonds, en lien avec la logique intuitionniste. Prenz par exemple le type du paragraphe précédent, imaginez que a, b, c sont des propositions logiques et que la flèche est le symbole d'implication. Nous obtenons une tautologie. Houla ! Ce n'est qu'un exemple de ce qui est appelé l'isomorphisme de Curry-Howard.

En cas de demandes pressantes et répétées, des notebooks sur le λ-calcul typé pourraient voir le jour ...

In [ ]:
```
```