Lambda Calcul

I. Termes, réduction, forme normale

Marc Lorenzi - 28 mai 2018

In [1]:
```
from lambda_parser import parse
from alias import *
```

Ce notebook est une introduction au λ-calcul pur non typé. Si, à tout hasard, un expert le lisait, il y trouverait sans doute des passages qu'il qualifierait de non rigoureux. Taille du notebook oblige, je devrai passer sous silence beaucoup de choses.

0. Introduction

Mea Culpa : Avant d'entrer dans le vif d'une sujet, une introduction un peu longuette est obligatoire. Soyez patient et attentifs, sinon la suite vous paraîtra incompréhensible.

considérons la fonction $f(x) = x - 1$. Votre prof de maths vous a sans doute dit que ceci n'est pas une fonction mais l'image de x par la fonction f. Et il a raison. La fonction c'est $f : x \mapsto x - 1$. Certains langages de programmation permettent de manipuler des fonctions sans les nommer. Pour n'en citer que quelques uns :

- Python : `lambda x: x - 1`
- Caml : `fun x -> x - 1`
- Haskell : `\x -> x - 1`
- Lisp : `(lambda (x) (- x 1))`

Étant donnés une fonction f et un objet x adéquat, on peut calculer l'image de x par f :

- Python `f(x)`
- Caml : `f x`
- Haskell : `f x`
- Lisp : `(f x)`

Raisonnons sur Python. Supposons que `f = lambda x: x - 1` . Comment calculer $f(2)$? On prend l'expression qui suit les deux points, $x - 1$, et on remplace x par 2. En Python, la notation `lambda` est pratique, mais un peu anecdotique. Mais dans des langages comme Haskell ou Caml, la notion de fonction est au coeur du système : ces langages sont appelés des **langages fonctionnels**.

Les langages fonctionnels sont bâtis à partir d'une théorie qui tente de construire le concept de fonction, non pas de façon abstraite, mais en tant que modèle permettant un calcul.

Cette théorie est le λ-calcul.

Qu'est-ce qu'une fonction calculable ? Dans la première moitié du XXe siècle, trois grands modèles ont vu le jour. Il s'agit :

- des machines de Turing, imaginées par ... Alan Turing.
- des fonctions récursives, créées par Kurt Gödel pour modéliser l'arithmétique des entiers.
- du λ-calcul, fondé par Alonzo Church.

Chacune de ces théories définissait ce qu'est une fonction "calculable". Ainsi, pour Turing, une fonction calculable est une fonction f pour laquelle il existe une machine de Turing telle que, pour tout x adéquat, si on donne x en entrée à la machine, celle ci fournit la valeur de $f(x)$.

Et alors ? Ces trois théories sont profondément différentes de prime abord. Et pourtant ... Soient $\mathcal{T}$ l'ensemble des fonctions calculables par une machine de Turing, $\mathcal{G}$ l'ensemble des fonctions récursives au sens de Gödel et $\mathcal{C}$ l'ensemble des fonctions calculables dans le λ-calcul.

Théorème : $\mathcal{T} = \mathcal{G} = \mathcal{C}$.

Ce résultat montrait que l'on tenait peut-être la réponse à la question : **Qu'est-ce que calculer ?** Il fut à l'origine de la **Thèse de Church**.

Thèse de Church : "Une fonction est calculable si et seulement si elle est calculable au sens de moi-même" (ou de Turing, ou de Gödel).

Nous allons développer dans ce notebook et ceux qui suivent les rudiments du λ-calcul et voir comment ce système, en apparence rudimentaire, est d'une puissance aussi grande que tous les systèmes de calculs qui ont pu être imaginés.

1. Syntaxe

1.1 Fonction ? Vous avez dit fonction ?

Qu'est-ce qu'une fonction ? Tout le monde le sait, c'est un triplet $(E, F, \mathcal{G})$ tel que $\mathcal{G} \subset E \times F$ et vérifiant de plus $\forall x \in E, \exists ! y \in F, (x, y) \in \mathcal{G}$. Enfin, c'est ce que vous a dit votre prof de maths et c'est vrai.

D'une certaine façon. Parce que juste avant votre premier vrai cours sur les applications, vous pensiez qu'une fonction de E vers F c'est une "façon" "d'associer" à tout élément $x \in E$ un unique élément $y \in F$, calculé à partir de x. Ce qui n'a rien à voir avec la définition académique.

Peut-on mathématiser VOTRE façon de voir ? Oui, on le peut. Le λ-calcul a été développé dans les années 1930 par Alonzo Church. Le λ-calcul a un très grand nombre d'applications. Pour ne parler que d'informatique, il est par exemple à la base des langages de programmation fonctionnels comme **Lisp**, **Haskell**, **Erlang** ou **Ocaml**. ET de nombreux autres langages (**Java**, **Python**, **C#**, ...) ont emprunté les idées du λ-calcul pour enrichir leurs constructions syntaxiques et simplifier grandement le travail du programmeur. Le λ-calcul est également utilisé pour étudier la **sémantique** des langages.

1.2 Les λ-termes

On suppose donné un ensemble $\mathcal{V} = \{v_0, v_1, v_2, \dots\}$ dont les éléments seront appelés des variables. Peu importe ce que sont les variables, des animaux de la forêt, des fleurs des champs, des entiers naturels ... Nous les noterons par des lettres minuscules $x, y, a, b,$ etc.

Définition :

- Une **variable** est un terme.
- Si M et N sont des termes, (MN) est un terme. On appelle ce type de terme une **application**.
- Si x est une variable est M est un terme, $(\lambda x.\, M)$ est un terme. On appelle ce type de terme une **abstraction**.

Il y a donc trois sortes de termes : les variables, les applications et les abstractions. En gros :

- Les variables sont les objets sur lesquels on applique des fonctions.
- Les abstractions sont les fonctions.
- Les applications sont les images des objets par des fonctions.

Par exemple $((\lambda x.\, xy)a)$ est un terme. Il représente (enfin il devrait) l'image de a par la fonction qui à x associe xy. Mais xy c'est quoi ? Eh bien c'est l'image de y par la fonction x.

Nous allons faire quelques conventions pour éviter les pléthores de parenthèses :

- $ABCD$ représente $(((AB)C)D)$. Dit autrement, l'application associe à gauche.
- Le "champ" (scope en anglais) de $\lambda x.\, \dots$ s'étend aussi loin à droite que possible. Ainsi, $\lambda x.\, xy$ représente $(\lambda x.\, (xy))$. Mais $(\lambda x.\, x)y$ représente $((\lambda x.\, x)y)$.

Au risque de me répéter, quel est le sens de tout cela ? Intuitivement, en λ-calcul, *tout est fonction*.

- La variable x représente une fonction "quelconque".
- Le terme $\lambda x.\, M$ représente la fonction qui à x associe M. Pensez à l'instruction Python `lambda x: x + 2` , c'est exactement cela.
- L'abstraction MN représente l'image de N par la fonction M.

Par exemple, $(\lambda x.\, x)y$ représente l'image de y par la fonction qui à x associe x. Évidemment, on aimerait que cela soit égal à y. Et ce le sera bientôt, mais patience ...

1.3 Représenter les termes en Python

Dans tout projet, le choix de la façon de représenter les objets à manipuler dans notre langage préféré est crucial. Une implémentation très propre du λ-calcul utiliserait sans aucun doute des classes Python, mais j'y vois deux inconvénients majeurs :

- Les classes ne sont pas au programme de l'Informatique Pour Tous en classes prépas.
- Elles se prêtent très mal à une découverte du code petit à petit. On définit TOUTE la classe en un seul bloc.

Bref, nous allons représenter un terme en Python par la liste de ses "constituants", augmentée d'un drapeau qui précise son type. Soyons plus précis :

- La variable x est représentée par la liste $[0, x]$.
- L'application MN est représentée par la liste $[1, M, N]$.
- L'abstraction $\lambda x.\, M$ est représentée par la liste $[2, x, M]$.

Pour nous simplifier la vie, nous nous autoriserons également à'utiliser dans les termes des "alias". Un alias est un nom en majuscules, censé représenter un raccourci pour un terme. Par exemple, si K est un alias représentant le terme $\lambda x.\, \lambda y.\, x$, Kab est le terme $(\lambda x.\, \lambda y.\, x)ab$.

- L'alias A sera représenté par la liste $[3, A]$.

Notons bien que les alias ne font pas partie du langage des termes. Avant d'effectuer des opérations sur les termes nous devrons en faire disparaître les alias.

Voici trois fonctions pour créer des termes.

In [2]:
```
def new_var(x): return [0, x]
def new_app(M, N): return [1, M, N]
def new_abs(x, M): return [2, x, M]
```

In [3]:
```
new_abs('x', new_app(new_var('x'), new_var('y')))
```

Out[3]:
```
[2, 'x', [1, [0, 'x'], [0, 'y']]]
```

La fonction `prod` prend en paramètre une liste $[M_1, \ldots, M_n]$ de termes. Elle renvoie le terme $M_1 \ldots M_n$.

In [4]:
```
def prod(ts):
    n = len(ts)
    if n == 0:
        raise Exception('Empty List of Terms')
    else:
        t = ts[0]
        for k in range(1, n):
            t = new_app(t, ts[k])
        return t
```

In [5]:
```
T = prod([new_var('x'), new_var('y'), new_var('z')])
print(T)
```

```
[1, [1, [0, 'x'], [0, 'y']], [0, 'z']]
```

Les fonctions ci-dessous permettent de tester le "type" d'un terme.

In [6]:
```
def is_var(P): return P[0] == 0
def is_app(P): return P[0] == 1
def is_abs(P): return P[0] == 2
def is_alias(P): return P[0] == 3
```

In [7]:
```
is_var(T), is_app(T), is_abs(T), is_alias(T)
```

Out[7]:
```
(False, True, False, False)
```

Et les suivantes permettent d'extraire les composantes d'un terme.

In [8]:
```
def var(P):
    if is_var(P) or is_abs(P) or is_alias(P): return P[1]
    else:
        raise Exception('Applications have no variable')
```

In [9]:
```
def left(P):
    if is_app(P): return P[1]
    elif is_var(P) or is_alias(P):
        raise Exception('Variables and aliases have no left field')
    else:
        raise Exception('Abstractions have no left field')

def right(P):
    if is_app(P): return P[2]
    elif is_var(P) or is_alias(P):
        raise Exception('Variables and aliases have no right field')
    else:
        raise Exception('Abstractions have no right field')
```

In [10]:
```
def expr(P):
    if is_abs(P): return P[2]
    elif is_var(P) or is_alias(P):
        raise Exception('Variables and aliases have no expr field')
    else:
        raise Exception('Applications have no expr field')
```

In [11]:
```
left(T)
```

Out[11]:
```
[1, [0, 'x'], [0, 'y']]
```

In [12]:
```
right(T)
```

Out[12]:
```
[0, 'z']
```

1.4 Un analyseur syntaxique

Le module `lambda_parser` contient une fonction `parse` . Celle ci a été importée au début du notebook. Elle prend en paramètre une chaîne de caractères représentant un λ-terme et renvoie la représentation Python de ce terme. Comme vous avez sans doute remarqué que votre clavier ne comporte pas de touche λ, j'ai choisi tout à fait arbitrairement le caracère `#` pour le remplacer.

Testez-la sur quelques exemples. Cette fonction détecte mal certaines erreurs de syntaxes (j'y travaille, inutile d'appeler un avocat), mais elle suffira pour nos besoins.

In [13]:
```
parse('xyz')
```

Out[13]:
```
[1, [1, [0, 'x'], [0, 'y']], [0, 'z']]
```

In [14]:
```
parse('x(yz)')
```

Out[14]:
```
[1, [0, 'x'], [1, [0, 'y'], [0, 'z']]]
```

In [15]:
```
parse('(#x.yz)(#t.x)a')
```

Out[15]:
```
[1, [1, [2, 'x', [1, [0, 'y'], [0, 'z']]], [2, 't', [0, 'x']]], [0,
'a']]
```

Un dernier exemple. Tiens, on dirait une ligne de programme ... Ce n'est pas une coïncidence, mais chaque chose en son temps.

In [16]:
```
parse('IF (ZERO x)(ADD x p)(MUL x q)')
```

Out[16]:
```
[1,
 [1,
  [1, [3, 'IF'], [1, [3, 'ZERO'], [0, 'x']]],
  [1, [1, [3, 'ADD'], [0, 'x']], [0, 'p']]],
 [1, [1, [3, 'MUL'], [0, 'x']], [0, 'q']]]
```

1.5 Conversion en chaîne de caractères

La représentation des termes en Python n'est pas très lisible. Voici de quoi rendre les choses plus claires. La fonction `to_string` prend un terme en paramètre et renvoie une représentation de celui-ci sous forme de chaîne de caractères. Notez le `u'\u03BB'` à la dernière ligne : c'est le code Unicode du caractère λ.

Rien de particulier dans le code de cette fonction, on essaie juste de mettre les parenthèses nécessaires et suffisantes.

In [17]:
```
def to_string(P):
    if is_var(P): return var(P)
    elif is_alias(P): return var(P) + ' '
    elif is_app(P):
        M = left(P)
        N = right(P)
        s1 = to_string(M)
        s2 = to_string(N)
        if is_abs(M):
            s1 = '(' + s1 + ')'
        if is_app(N)or is_abs(N):
            s2 = '(' + s2 + ')'
        return s1 + s2
    else:
        return u'\u03BB' + var(P) + '.' + to_string(expr(P))
```

In [18]:
```
to_string(parse('(#x.yz)(#t.x)(a)'))
```

Out[18]:
```
'(λx.yz)(λt.x)a'
```

Super, ça enlève même les parenthèses inutiles.

In [19]:
```
def print_term(t):
    print(to_string(t))
```

In [20]:
```
print_term(parse('#x.#y.z(#t.tu)x'))
```

```
λx.λy.z(λt.tu)x
```

Voulez-vous voir tous les alias définis jusqu'à présent ?

In [21]:
```
def print_aliases():
    for A in aliases:
        print(A, to_string(aliases[A]))
```

In [22]:
```
print_aliases()
```

```
I λx.x
S λf.λg.λx.fx(gx)
K λa.λx.a
TEST S K K
OMEGA (λx.xx)(λx.xx)
```

1.6 Macro-expansion

Les alias ne font pas partie du langage des termes, ils ne sont que des raccourcis. Un "vrai" terme ne contient pas d'alias. La fonction `expand1` prend un "faux" terme en paramètre et remplace tous les alias par leur valeur. Elle renvoie un couple (T, b), où T est le terme obtenu et b est un booléen valant `True` s'il y a eu au moins un remplacement.

À quoi sert b ? Bientôt la réponse.

In [23]:
```
def expand1(P):
    if is_var(P): return (P, False)
    elif is_alias(P): return (aliases[var(P)], True)
    elif is_app(P):
        M, b1 = expand1(left(P))
        N, b2 = expand1(right(P))
        return (new_app(M, N), b1 or b2)
    else:
        M, b = expand1(expr(P))
        x = var(P)
        return (new_abs(x, M), b)
```

Par exemple, l'alias `TEST` est défini comme étant SKK, où S et K sont eux-mêmes des alias.

In [24]:
```
print_term(parse('TEST'))
```

```
TEST
```

In [25]:
```
T, b = expand1(parse('TEST'))
print_term(T)
```

```
S K K
```

Ah mais oui, un alias peut contenir des alias, qui contiennent des alias ... il faut donc appeler `expand1` autant de fois que nécessaire pour que tous les alias disparaissent. D'où l'utilité du booléen b.

Remarque : il est de votre responsabilité d'éviter les cercles vicieux : les alias **ne peuvent pas** être récursifs.

In [26]:
```
def expand(P):
    while True:
        P, b = expand1(P)
        if not b: break
    return P
```

In [27]:
```
print_term(expand(parse('S')))
print_term(expand(parse('K')))
print_term(expand(parse('TEST')))
```

```
λf.λg.λx.fx(gx)
λa.λx.a
(λf.λg.λx.fx(gx))(λa.λx.a)(λa.λx.a)
```

Comment créer un alias ? Rien de plus facile, la fonction `add_alias` du module `alias` a été importée. Allez lire le code du fichier `alias.py` , il fait une dizaine de lignes.

In [28]:
```
add_alias('COMP', '#g.#f.#x.g(f x)')
```

In [29]:
```
print_term(expand(parse('COMP')))
```

```
λg.λf.λx.g(fx)
```

Exercice : pourquoi `COMP` ? Qu'est censé modéliser ce terme ?

Remarquez que nos alias s'enrichissent au fil du notebook.

In [30]:
```
print_aliases()
```

```
I λx.x
S λf.λg.λx.fx(gx)
K λa.λx.a
TEST S K K
OMEGA (λx.xx)(λx.xx)
COMP λg.λf.λx.g(fx)
```

2. Variables libres, variables liantes, variables liées

Définition : Soit T un terme. Soit x une variable.

- On dit qu'une occurence de x est **liante** dans T si elle apparaît juste après un λ.
- On dit qu'une occurence de x est **liée** dans T si elle apparaît dans le M d'un "sous-terme" de T de la forme $\lambda x.\, M$.
- Sinon on dit que l'occurence de x est **libre**.

Je ne définis pas le mot **occurence**. Disons qu'une occurence de x c'est un x bien précis parmi tous les x du terme considéré.

Par exemple, dans le terme $\lambda x.\, xy$, la première occurence de x est liante, la seconde est liée, et l'unique occurence de y est libre.

Regardons un exemple un peu plus subtil. Considérons le terme $T = (\lambda x.\, \lambda y.\, xz)x(\lambda x.\, x)$. Ce terme contient les trois variables x, y et z.

- x a deux occurences liantes, deux occurences liées, et une occurence libre.
- z a une seule occurence qui est libre.
- y a une occurence liante et aucune occurence liée.

Aparté : le lecteur ayant une certaine maturité mathématique sait que le nom choisi pour une variable liée importe peu. Ainsi, $\int_0^1 x^3\,dx = \int_0^1 t^3\,dt$, etc.

Il en va de même en λ-calcul. Considérons les termes $M = \lambda x.\, xy$ et $N = \lambda t.\, ty$. Bien que ce ne soient pas les mêmes termes, nous les considérerons comme égaux. Une approche plus rigoureuse consisterait à définir une relation d'équivalence sur les termes, l'égalité à "renommage près des variables liées". Pour les experts, cette relation s'appelle l'α-équivalence. Nous n'en parlerons plus et nous considérerons deux tels termes comme égaux.

Notons pour tout terme T par $FV(T)$ l'ensemble des variables libres de T (Free Variables). On a

- $FV(x) = \{x\}$.
- $FV(MN) = FV(M) \cup FV(N)$.
- $FV(\lambda x.\, M) = FV(M) \setminus \{x\}$

Exercice : Définissez $BV(T)$ (B pour bound).

Il est facile d'écrire des fonctions qui décident si une variable a une occurence libre (resp. liante, liée) dans un terme. Il suffit d'appliquer la définition.

In [31]:
```
def is_free(x, P):
    if is_var(P): return x == var(P)
    elif is_app(P):
        return is_free(x, left(P)) or is_free(x, right(P))
    else:
        return x != var(P) and is_free(x, expr(P))
```

In [32]:
```
T = parse('(#x.#y.xz)x(#x.x)')
print_term(T)
```

```
(λx.λy.xz)x(λx.x)
```

In [33]:
```
print(is_free('x', T), is_free('y', T), is_free('z', T))
```

```
True False True
```

In [34]:
```
def is_binding(x, P):
    if is_var(P): return False
    elif is_app(P):
        return is_binding(x, left(P)) or is_binding(x, right(P))
    else:
        return x == var(P) or is_binding(x, expr(P))
```

In [35]:
```
print(is_binding('x', T), is_binding('y', T), is_binding('z', T))
```

```
True True False
```

In [36]:
```
def is_bound(x, P):
    if is_var(P): return False
    elif is_app(P):
        return is_bound(x, left(P)) or is_bound(x, right(P))
    else:
        Q = expr(P)
        return (x == var(P) and is_free(x, Q)) or is_bound(x, Q)
```

In [37]:
```
print(is_bound('x', T), is_bound('y', T), is_bound('z', T))
```

```
True False False
```

Remarquez que y, bien que liante, n'est pas liée dans T car $\lambda y.\, xz$ est "constante en y".

Il est tout aussi facile d'écrire des fonctions qui renvoient l'ensemble des variables libres (resp. liées, liantes) d'un terme.

In [38]:
```
def free_vars(P):
    if is_var(P): return set([var(P)])
    elif is_app(P):
        return free_vars(left(P)).union(free_vars(right(P)))
    else:
        return free_vars(expr(P)).difference(set([var(P)]))
```

In [39]:
```
free_vars(T)
```

Out[39]:
```
{'x', 'z'}
```

In [40]:
```
def bound_vars(P):
    if is_var(P): return set([])
    elif is_app(P):
        return bound_vars(left(P)).union(bound_vars(right(P)))
    else:
        Q = expr(P)
        B = bound_vars(Q)
        x = var(P)
        if is_free(x, Q): return B.union(set([x]))
        else: return B
```

In [41]:
```
bound_vars(T)
```

Out[41]:
```
{'x'}
```

In [42]:
```
print(free_vars(expand(parse('COMP'))))
print(bound_vars(expand(parse('COMP'))))
```

```
set()
{'x', 'g', 'f'}
```

Le terme `COMP` n'a pas de variables libres. On dit que c'est un terme clos, ou un **combinateur**.

In [43]:
```
def binding_vars(P):
    if is_var(P): return set([])
    elif is_app(P):
        return binding_vars(left(P)).union(binding_vars(right(P)))
    else:
        return binding_vars(expr(P)).union(set([var(P)]))
```

In [44]:
```
binding_vars(T)
```

Out[44]:
```
{'x', 'y'}
```

3. Substitution

Soient M, N deux termes et x une variable. Nous voudrions remplacer chaque occurence ***libre*** de x dans M par le terme N. Nous notons $M[x := N]$ le terme obtenu.

Cela semble très facile : dans la définition ci-dessous, les lettres minuscules représentent des variables, les lettres majuscules sont des termes quelconques.

Définition fausse :

1. $y[x := N] = N$ si $x = y$.
2. $y[x := N] = y$ si $x \neq y$.
3. $(PQ)[x := N] = (P[x := N])(Q[x := N])$.
4. $(\lambda y.\, P)[x := N] = \lambda y.\, (P[x := N])$ <-- **NON !**

Cette définition pose problème.

Les cas 1, 2 et 3 sont corrects. Mais considérons le terme $T = \lambda y.\, x$. Moralement, c'est la fonction constante égale à x. Que vaut $T[x := w]$? Selon notre définition, c'est $\lambda y.\, w$. Tout va bien, c'est la fonction constante égale à w. Maintenant, selon notre définition, que vaut $T[x := y]$? Cela devrait être la fonction constante égale à y. Malheureusement cela fait $\lambda y.\, y$, la fonction identité, pas du tout constante. Que s'est il passé ? La variable libre y a été capturée par la variable liante du même nom.

Une idée ? Oui, il faut changer le nom de la variable liante avant d'effectuer la substitution. Prendre une variable z toute neuve, et calculer $T[y := z][x := y]$. Cette fois ci, on obtient $\lambda z.\, y$, la fonction constante égale à y.

Voici donc la définition définitive.

Définition :

1. $y[x := N] = N$ si $x = y$.
2. $y[x := N] = y$ si $x \neq y$.
3. $(PQ)[x := N] = (P[x := N])(Q[x := N])$.
4. $(\lambda x.\, P)[x := N] = \lambda x.\, P$.
5. Si $x \neq y$ et $x \notin FV(P)$, $(\lambda y.\, P)[x := N] = \lambda y.\, P$.
6. Si $x \neq y$, $x \in FV(P)$ et $y \notin FV(N)$, $(\lambda y.\, P)[x := N] = \lambda y.\, (P[x := N])$.
7. Si $x \neq y$, $x \in FV(P)$ et $y \in FV(N)$, $(\lambda y.\, P)[x := N] = \lambda z.\, (P[y := z][x := N])$ où z est une "nouvelle variable".

La substitution est une notion délicate, à cause du problème de la capture des variables. Soyez courageux, une fois ce problème réglé le reste sera beaucoup plus facile.

Question programmation, la condition numéro 6 fait un peu peur. Comment créer une "nouvelle variable" ? Une fois n'est pas coutume, écrivons une classe `VarProvider`. Un objet de cette classe est un fournisseur de variable. Le mieux est de voir tout cela fonctionner sur un exemple.

In [45]:
```
class VarProvider:

    def __init__(self, prefix='v'):
        self.prefix = prefix
        self.count = 0

    def reset(self): self.count = 0

    def fresh_var(self):
        self.count += 1
        return self.prefix + str(self.count)
```

In [46]:
```
vp = VarProvider('z')
print(vp.fresh_var(), vp.fresh_var(), vp.fresh_var())
vp.reset()
print(vp.fresh_var(), vp.fresh_var())
```

```
z1 z2 z3
z1 z2
```

Chaque appel à la méthode `fresh_var` renvoie une nouvelle variable. Ce n'était pas bien compliqué.

Voici donc la fonction de substitution. Elle reprend point par point la définition de la substitution, avec les mêmes notations. Elle prend 4 paramètres : un terme M, une variable x, un terme N et un `VarProvider` vp. La fonction renvoie $M[x := N]$.

In [47]:
```
def subs(M, x, N, vp):
    if is_var(M):
        y = var(M)
        if x == y: return N                          # Cas 1
        else: return M                               # Cas 2
    elif is_app(M):                                  # Cas 3
        P = left(M)
        Q = right(M)
        return new_app(subs(P, x, N, vp), subs(Q, x, N, vp))
    else:
        y = var(M)
        P = expr(M)
        if x == y: return M                          # Cas 4
        # Maintenant, x != y
        elif not(is_free(x, P)): return M            # Cas 5
        # Maintenant, x est libre dans P
        elif not(is_free(y, N)):                     # Cas 6
            return new_abs(y, subs(P, x, N, vp))
        else:                                        # Cas 7, le plus co
            z = vp.fresh_var()
            P1 = subs(P, y, new_var(z), vp)
            P2 = subs(P1, x, N, vp)
            return new_abs(z, P2)
```

Un exemple ?

In [48]:
```
T = parse('(#x.#y.xyz)a(#x.y)')
print_term(T)
print_term(subs(T, 'y', parse('x'), VarProvider()))
```

```
(λx.λy.xyz)a(λx.y)
(λx.λy.xyz)a(λv1.x)
```

4. β-réduction

Dans λ-calcul il y a λ. Mais il y a aussi "calcul". Alors quand est-ce qu'on va calculer quelque chose ?

Imaginons un terme du genre $(\lambda x.\, M)N$. Intuitivement ce terme représente l'image de N par la fonction $x \mapsto M$. Comment calculer cette image ? Eh bien on remplace tous les x qui sont dans M par des N. Mais on sait faire, c'est une substitution ! Le λ-calcul (disons celui que je vous expose ici) possède une unique règle, appelée la règle de β-réduction.

Règle β : $(\lambda x.\, M)N \rightarrow_\beta M[x := N]$**.**

Un terme du genre $(\lambda x.\, M)N$ est ce que l'on appelle un **redex**. Calculer, c'est faire des β-réductions tant qu'il y a des redex, jusqu'à aboutir à un terme qui n'a plus de redex. Un tel terme est appelé une forme normale, c'est le résultat du calcul.

Calculer, c'est réduire à une forme normale.

La fonction `beta_reduce` prend un terme en paramètre. Elle cherche un redex dans ce terme. Si elle en trouve un elle effectue la β-reduction de ce redex et renvoie le nouveau terme et le booleen `True` .

Sinon, elle renvoie le terme passé en paramètre et le booléen `False` .

In [49]:
```
def beta_reduce(P):
    if is_var(P):
        return (P, False)
    elif is_app(P):
        if is_abs(left(P)):
            x = var(left(P))
            A = expr(left(P))
            vp = VarProvider()
            return (subs(A, x, right(P), vp), True)
        else:
            P1, b = beta_reduce(left(P))
            if b: return (new_app(P1, right(P)), True)
            else:
                Q1, b1 = beta_reduce(right(P))
                return (new_app(left(P), Q1), b1)
    else:
        Q, b = beta_reduce(expr(P))
        return (new_abs(var(P), Q), b)
```

Comment la fonction `beta_reduce` ci-dessus trouve-t-elle un redex dans le terme M ? Récursivement, cela va de soi.

- Si $M = \lambda x.P$ est une abstraction, on cherche un redex dans P.
- Si $M = PQ$ est une application :
  - Si M est un redex (c'est à dire si P est une abstraction), bravo, on a trouvé ! Sinon :
  - On cherche un redex dans P
  - Si on n'en trouve pas, on cherche un redex dans Q

Pour dire les choses autrement, la fonction effectue la réduction du redex **le plus à gauche** de M, si ce redex existe.

In [50]:
```
T, b = beta_reduce(parse('(#x.xxx)(ab)'))
print_term(T)
```

```
ab(ab)(ab)
```

5. Forme normale

5.1 C'est quoi ?

Definition :

- Un terme **est** une forme normale s'il ne contient pas de redex.
- Un terme **a** une forme normale s'il existe une suite de β-réductions qui transforment ce terme en une forme normale.

Voici donc la fonction qui calcule une forme normale d'un terme P.

In [51]:
```
def normal_form(P):
    b = True
    while b:
        P, b = beta_reduce(P)
    return P
```

En voici une version un peu plus sophistiquée :

Le paramètre `max_red` assure la terminaison. Si après `max_red` réductions aucune forme normale n'est trouvée, la fonction s'arrête et renvoie le résultat obtenu à ce moment-là.

Le paramètre `trace` permet l'affichage ou pas des résultats intermédiaires et du nombre de réductions.

In [52]:
```
def normal_form(P, max_red=1000, trace=True):
    b = True
    count = 0
    if trace: print(count, to_string(P))
    while b and count < max_red:
        P, b = beta_reduce(P)
        if b: count += 1
        if trace and b: print(count, to_string(P))
    if trace:
        print(20 * '-')
        print('%d réductions\n' % count)
    return P
```

In [53]:
```
T = parse('(#x.x)y')
normal_form(T)
```

```
0 (λx.x)y
1 y
--------------------
1 réductions
```

Out[53]:
```
[0, 'y']
```

Ouf, c'est bien ce que l'on voulait ! Prenons quelques autres exemples.

In [54]:
```
T = parse('(#x.#y.#z.#t.xyzt)abc')
normal_form(T)
```

```
0 (λx.λy.λz.λt.xyzt)abc
1 (λy.λz.λt.ayzt)bc
2 (λz.λt.abzt)c
3 λt.abct
--------------------
3 réductions
```

Out[54]:
```
[2, 't', [1, [1, [1, [0, 'a'], [0, 'b']], [0, 'c']], [0, 't']]]
```

In [55]:
```
T = expand(parse('TEST'))
normal_form(T)
```

```
0 (λf.λg.λx.fx(gx))(λa.λx.a)(λa.λx.a)
1 (λg.λx.(λa.λx.a)x(gx))(λa.λx.a)
2 λx.(λa.λx.a)x((λa.λx.a)x)
3 λx.(λv1.x)((λa.λx.a)x)
4 λx.x
--------------------
4 réductions
```

Out[55]:
```
[2, 'x', [0, 'x']]
```

Théorème : $SKK \to_\beta \ldots \to_\beta I$ où $I = \lambda x.\, x$.

5.2 Unicité de la forme normale

Définition Étant donnés deux termes M et N, $M \to_\beta^* N$ (noter l'étoile) lorsqu'on peut passer de M à N par une suite de 0, une ou plusieurs β-réductions. On définit ainsi une relation sur l'ensemble des termes, qui est clairement réflexive et transitive. C'est en fait la plus petite relation réflexive et transitive contenant $\to_\beta$.

Cette relation n'est en revanche pas antisymétrique. C'est ce que l'on appelle un pré-ordre. Dans l'exemple ci-dessous nous voyons apparaître deux termes distincts M et N tels que $M \to_\beta^* N$ et $N \to_\beta^* M$.

In [56]:
```
T = expand(parse('(#x.I xx)(#x.I xx)'))
normal_form(T,max_red=5)
```

```
0 (λx.(λx.x)xx)(λx.(λx.x)xx)
1 (λx.x)(λx.(λx.x)xx)(λx.(λx.x)xx)
2 (λx.(λx.x)xx)(λx.(λx.x)xx)
3 (λx.x)(λx.(λx.x)xx)(λx.(λx.x)xx)
4 (λx.(λx.x)xx)(λx.(λx.x)xx)
5 (λx.x)(λx.(λx.x)xx)(λx.(λx.x)xx)
--------------------
5 réductions
```

Out[56]:
```
[1,
 [1,
  [2, 'x', [0, 'x']],
  [2, 'x', [1, [1, [2, 'x', [0, 'x']], [0, 'x']], [0, 'x']]]],
 [2, 'x', [1, [1, [2, 'x', [0, 'x']], [0, 'x']], [0, 'x']]]]
```

La relation $\rightarrow_\beta^*$ possède une propriété fondamentale :

Théorème de Church-Rosser : Soient M, N_1, N_2 trois termes tels que $M \rightarrow_\beta^* N_1$ et $M \rightarrow_\beta^* N_2$. Alors, il existe un terme T tel que $N_1 \rightarrow_\beta^* T$ et $N_2 \rightarrow_\beta^* T$.

Dit autrement, si on effectue deux calculs "divergents" à partir d'un terme M en choisissant des réductions différentes, il y a moyen de faire re-converger les calculs vers un même résultat.

Corollaire : Si un terme M a une forne normale, celle-ci est unique.

Démonstration : Supposons que le terme M ait N_1 et N_2 pour formes normales. Par le théorème de Church-Rosser, il existe un terme T tel que $N_1 \rightarrow_\beta^* T$ et $N_2 \rightarrow_\beta^* T$. Mais les N_i sont normales, elles n'ont pas de rédex. Il n'existe aucune β-réduction de celles-ci. Donc, $N_1 = N_2 = T$.

5.3 Existence de la forme normale ?

Tout le monde l'attend, ce théorème d'existence ! Eh bien il n'existe pas. Il y a des termes qui n'ont pas de forme normale. En voici un :

In [57]:
```
T = expand(parse('OMEGA'))
normal_form(T,max_red=10)
```

```
0 (λx.xx)(λx.xx)
1 (λx.xx)(λx.xx)
2 (λx.xx)(λx.xx)
3 (λx.xx)(λx.xx)
4 (λx.xx)(λx.xx)
5 (λx.xx)(λx.xx)
6 (λx.xx)(λx.xx)
7 (λx.xx)(λx.xx)
8 (λx.xx)(λx.xx)
9 (λx.xx)(λx.xx)
10 (λx.xx)(λx.xx)
--------------------
10 réductions
```

Out[57]:
```
[1, [2, 'x', [1, [0, 'x'], [0, 'x']]], [2, 'x', [1, [0, 'x'], [0, 'x
']]]]
```

Le terme $\Omega = (\lambda x.\, xx)(\lambda x.\, xx)$ est un redex, il n'est donc pas une forme normale. La seule réduction possible est $\Omega \rightarrow_\beta \Omega$. Peine perdue.

5.3 Tout est-il perdu ?

Soit T un terme. Peut-on prouver que T a (ou n'a pas) une forme normale ?

Théorème : Il n'existe aucun algorithme qui décide, en un temps fini, si un terme donné a ou non une forme normale.

La situation s'aggrave. Il y a pire. Soit $T = Ka\Omega$ où $K = \lambda x.\, \lambda y.\, x$. D'un côté, on a $T \to_\beta^* T \to_\beta^* T \ldots$ si on passe son temps à faire des β-réductions sur Ω. Ce terme n'a-t-il donc pas de forme normale ? Faux ! en effet,

$$T = (\lambda x.\, \lambda y.\, x)a\Omega \to_\beta (\lambda y.\, x)[x := a]\Omega = (\lambda y.\, a)\Omega \to_\beta a[y := \Omega] = a$$

La forme normale de T est donc a. Il existe donc de "mauvaises" stratégies de réduction qui ne finissent jamais, même si nos terme ont une forme normale. Tout ce que prédit le théorème de Church-Rosser c'est que SI une stratégie de réduction termine, alors elle termine sur la forme normale.

Tout est-il perdu ? Non.

Théorème : Soit T un terme. Si T a une forme normale, alors la stratégie de réduction du redex "le plus à gauche" utilisée plus haut termine sur la forme normale de T.

On a eu de la chance : on a justement choisi la bonne stratégie :-).

6. Et maintenant ?

Reste-t-il un lecteur ou deux ? Bravo ! Il est vrai que tout ce que nous avons abordé jusqu'à maintenant demeure très abstrait, on ne voit pas bien à quoi cela peut servir. Patience. Dans les prochains numéros nous verrons que l'on peut à l'intérieur même du λ-calcul :

- Implémenter l'arithmétique des entiers naturels. Nombres, opérations $+$, $-$, $\times$, etc.
- Définir les booléens, les tests.
- Définir les couples, les triplets, ..., les tableaux.
- Programmer. Écrire des boucles, des fonctions récursives, etc.

Accrochez-vous : dans les prochains épisodes l'insoupçonnable va arriver.

In [ ]: