Interface PM
IFT2905
Feuille de notes

Franz Girardin

21 mai 2024

Table des matières

Style de language de programmation

- **Impératif**
  - Procédural
  - Objet
- **Déclaratif**
  - Fonctionnel
  - Logique
- **Concurrent**
  - Mémoire partagée
  - Passage de message

Le **niveau d'abstraction** d'un language est la distance conceptuelle dudit language par rapport au language machine.

Programmation impérative procédurale

- `Fortran`, `Algol 60`, `Pascal`, `C`, `Ada`

Effectuent des opérations sur la mémoire ; les instructions sont regroupées en *procédures*. Peut facilement être traduit (compilé) en language machine.

Programmation impérative OO

- `Simula`, `Smalltalk`, `C++`, `Java`

Chaque objet de la mémoire est accompagné de code qui lui permet d'interagir avec les autres objets. Les *méthodes* remplacent les *procédures*. Le **flot de contrôle** passe d'un objet à l'autre par appel de méthode.

Programmation Fonctionnelle

- `Lisp`, `ML`, `Haskell`, `APL`

Une fonction est un calcul. Facilite l'analyse et le raisonnement ; limite les effets de bord. Généralement apprécié pour son élégence.

Section 1

Syntaxe des expressions

La notation **infixée** est plus familière et intuitive mais elle peut aussi être embigue.

- **Niveau de précédence** $a + b * c \equiv a + (b * c)$
- **Associativité** $a - b - c \equiv (a - b) - c$

Les niveaux de précédences établisssent la priorité des opérations et associent les termes des expressions de la gauche vers la droite. Chaque langage peut avoir une grande quantité de niveaux et ils peuvent s'avérer difficile à mémoriser.

Définition formelle de la syntaxe Un **langage** est un ensemble de *phrases* qui sont composées de *séquences de symboles* ; cest symboles représentent le vocabulaire du language. La **grammaire** est l'ensemble des règles précisant l'usage du langage.

Language et grammaire Il est possible de définir une langage $L(G)$ à partir d'une grammaire G. L'ensemble $L(G)$ est l'ensemble des *chaînes* et des *phrases*qui peuvent être généré en utilisant les règles de grammaire G. Les éléments $L(G)$ sont noté p et représentent les phrases ou chaînes de caractèresproduites par la grammaire. L'expression `depart` $\Longrightarrow \cdots \Longrightarrow p$ signifie qu'il est possible d'utiliser un **symbole** initial et appliquer n'importe quelle série de règle de production de la grammaire G pour arriver à la chaîne p.

$$L(G) = \left\{ p \mid \langle \texttt{départ} \rangle \right\}$$

Backus-Naur Form La BNF est un système de notation pour décrire la syntaxe d'un langage sous forme de **règles de production**. Chaque règle décrit une structure syntaxique spécifique en termes de séquences d'autres structures, qui peuvent être des symboles terminaux (c'est-à-dire les éléments de base du langage, tels que des mots-clés, des opérateurs ou des identificateurs) ou d'autres structures syntaxiques définies par des règles.

Exemple 1

Pour définir une **catégorie** en BNF, on peut utiliser la syntaxe suivant

$$\langle \texttt{cat} \rangle ::= x_1, x_2, \cdots x_n$$

Pour définir un binaire, on peut utiliser la syntaxe suivante

$$\langle \texttt{bin} \rangle ::= 0$$
$$\langle \texttt{bin} \rangle ::= 1$$
$$\langle \texttt{bin} \rangle ::= \langle \texttt{bin} \rangle \langle \texttt{bin} \rangle$$

Définition 1 Dérivation directe

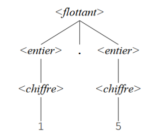
Il s'agit de l'application d'une **règle de production** définie en BNF Puisqu'il existe une règle définissant $\langle bin \rangle$, il est possible d'appliquer une dérivation directe de cette règle pour obtenir le binaire `01` :

$$\langle \texttt{bin} \rangle ::= \langle \texttt{bin} \rangle \langle \texttt{bin} \rangle \Longrightarrow \langle \texttt{bin} \rangle\ \texttt{0} \Longrightarrow \texttt{1 0}$$

Exemple 2 BNF pour type numérique

$$\langle \texttt{flottant} \rangle ::= \langle \texttt{entier} \rangle\ \texttt{.}\ \langle \texttt{entier} \rangle$$
$$\langle \texttt{entier} \rangle ::= \langle \texttt{chiffre} \rangle$$
$$\langle \texttt{entier} \rangle ::= \langle \texttt{chiffre} \rangle \langle \texttt{entier} \rangle$$
$$\langle \texttt{chiffre} \rangle ::= \langle \texttt{(0|1|2|3|4|5|6|7|8|9)} \rangle$$

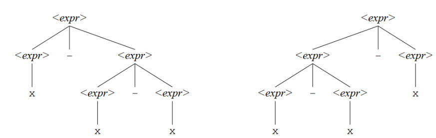
On peut vérifier que l'expression `1.5` est un flottant selon la définission BNF en observant l'**arbre de dérivation** suivant dans lequel le départ est la racine et chaque phrase est une feuille



Grammaires ambigues Une grammaire est **ambigue** ssi il existe une phrase dans $L(G)$ qui a plusieurs arbres de dérivations.

Exemple 3 Phrase ambigue

$$\langle \texttt{expr} \rangle ::= x$$
$$\langle \texttt{expr} \rangle ::= \langle \texttt{expr} \rangle - \langle \texttt{expr} \rangle$$



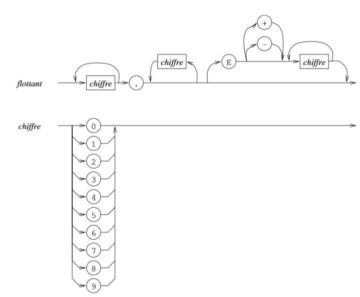
Autres motifs de syntaxes BNF

$x_1 | x_2$: *peut être*x_1**ou**x_2

(x) : *groupement*

$[x]$ **ou** $x?$ **ou** $\varepsilon | x$: x *peut apparaître* **0***ou***1** *fois*

$\{x\}$ **ou** : x *peut être répété* **0***ou* **plusieurs** *fois*



Sucre syntaxique

- Exension syntaxique superficielle **équivalente** à une autre syntaxe
- Pas d'impact sur les propriétés internes du langage

Langage fonctionnel Les langages **fonctionnels** fournissent une environnment pour générer du code à un haut niveau d'abstraction. Le programmation fonctionnelle est un paradigme de programmation pour laquelle les expressions sont plus importantes que les affirmations. On compose ainsi les programmes en utilisant des **expressions** ; chacune d'elles génère une valeur et ces expressions peuvent être combinées pour engendrer une expression plus complexes. Cette approche est différente de la programmation impérative où les affirmations ont un effet sur l'état global et où les affirmations communiquent des valeurs via l'état global.

Section **2**

Lambda calcul

Le calcul lambda décrit comment *définir* des fonctions et comment les appeler.

Syntaxe

$$\lambda x.x + 2$$

- ▷ λ : On déclare la **création** d'une fonction
- ▷ x : Déclare la **variable** dépendante de la f
- ▷ $.$: Sépare les déclarations de l'expression
- ▷ $x + 2$: Expression définissant f

$$(\lambda x.x + 2)\ 3 \leftrightarrow\ f(3)$$

- ▷ On appelle la f avec l'argument **3**
- ▶ Logiquement équivalent à $f(3)$

Concept. 1

Puisque les fonctions acceptent des **valeurs** ; on peut passer des fonctions comme valeur dans d'autres fonctions, comme lorsqu'on effectue une composition $f \circ g$

Définition 2 Termes lamda

Une **fonction** est une expression lamda ; elle est composée de trois **types** de *termes lambda* fondamentaux

Termes lambda

- ▷ **Variable**, p. ex. x
- ▷ **Fonction**, p. ex. $\lambda x.x + 2 \leftrightarrow f(x) = x + 2$
- ▷ **L'expression d'appel** ; *calling a function*
- ▶ p. ex $(\lambda x.x + 2)y \leftrightarrow f(y)$

Note :

L'utilisation de **parenthèse** dans $(\lambda x.x + 2)y$ permet d'identifier et de distinguer la variable (x) de l'expression de la fonction et la viriable (y) utilisée pour l'appel

Simplification d'expression Il existe des règles qui permettent de simplifier les expressions lambda et ainsi effectuer des calculs de fonction. Ces règles sont analogues aux règles utilisée pour simplifier des expressions algébriques

Règle 1

Le nom d'une fonction n'a pas d'importance.

$$\lambda x.x + 2 \leftrightarrow \lambda y.y + 2$$

Règle 2

On calcule une fonction en substituant la paramètre d'entrée définissant la fonction avec le terme d'entrée présenté pour le calcul

$$(\lambda x.x + 2)5 \leftrightarrow f(5) = 5 + 2$$

Syntaxe d'expression En lambda calcul, chaque composante d'une expression est aussi une expression (E). Et chaque expression doit satisfaire la syntaxe suivante.

$$\begin{aligned} &E \rightarrow \text{ID} \\ &E \rightarrow \lambda\ \text{ID}\ .\ E \\ &E \rightarrow E\ E \\ &E \rightarrow (E) \end{aligned}$$

Certaines expressions telles que $\lambda x.yz$ sont ambigue ; il faut donc des règles de disambiguation.

Règles de disambiguation

▷ $E \rightarrow E\ E$ est associatif à gauche

$$\begin{aligned} &xyz \leftrightarrow (xy)z \\ &wxyz \leftrightarrow ((wx)y)z \end{aligned}$$

▷ λ ID . E s'étend à droite autant que possible, commençant part λ ID

$$\begin{aligned} &\lambda x\ .\ xy \leftrightarrow \lambda x\ .\ (xy) \\ &\lambda x\ .\ \lambda x\ .\ x \leftrightarrow \lambda x\ .\ (\lambda x\ .\ x) \\ &(\lambda x\ .\ xy)z \leftrightarrow (\lambda x\ .\ (xy)) \\ &(\lambda x\ .\ wxy)z \leftrightarrow (\lambda x\ .\ ((wx)y))z \\ &\lambda x\ .\ (x) \leftrightarrow \lambda x\ .((x)y) \\ &\lambda a\ .\ \lambda b\ .\ \lambda c\ .\ abc \leftrightarrow \lambda a\ .\ (\lambda b\ .\ (\lambda c\ .\ ((ab)c))) \end{aligned}$$

Sémantique

▷ En lambda calcul, un ID tel que décrit par les règles de syntaxe est en fait une **variable**.

▷ $E \rightarrow \lambda$ ID . E est une **abstraction** et l'ID est la **variable** de l'abstraction. La seconde entité E est le **corps** de l'abstraction.

▷ $E \rightarrow E\ E$ est une **application**

▷ λ ID . E **définit** une nouvelle fonction anonyme

▷ $E \rightarrow E_1\ E_2$ correspond à **l'appel de la fonction** E_1 en établissant E_2 comme sont paramètre formel

Exemple 4 Fonction incrément

$$\lambda x\ .\ + x\ 1$$

Représente une fonctioon qui ajoute **1** à son argument.

$$(\lambda x\ .\ + x\ 1)\ 2$$

Représente l'appel ou *application* de cette fonction en établissant **2** comme étant paramètre formel.

$$(\lambda x\ .\ + x\ 1)\ 2 = 2 + 1 = 3$$

Or, la fonction suivante ne peut pas être calculée lorsqu'elle est appelé avec **2** :

$$\lambda y\ .\ + x\ 1$$

En effet, l'application de la fonction correspond à la substitution de tous les 'y' du corps de l'expression par le pamètre formel **2**. Mais le corps de l'expression définit un fonction en termes de x et non y ; la substitution ne peut donc pas se faire et l'évaluation de la fonction avec **2** se réduit uniquement à $x + 1$:

$$(\lambda x\ .\ + x\ 1)\ 2 = x + 1$$

Currying Il s'agit d'une technique pour traduire l'évaluation d'une fonction qui prend multiples arguments en une séquence de fonctions qui prennent chacune un **unique** argument.

Exemple 5

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

Est composé de deux **abstractions**. L'abstraction externe introduite par λx suggère qu'on doit substituer le paramètre formel là où le reste de l'expression contient la variable x. Ainsi, l'appel de fonction suivant peut être réduit :

$$\big(\lambda x\ .\ \lambda y\ .\ ((+x)y)\big)1 = \lambda y\ .\ ((+\ 1)y)$$

Exemple 6 Addition par currying

$$\begin{aligned} \big(\lambda x\ .\ \lambda y\ .\ ((+\ x)\ y)\big)\ 10\ 20 &= \big(\lambda y\ .\ ((+\ 10))\ y\big)\ 20 \\ &= (+\ 10)\ 20 \\ &= 10 + 20 \\ &= 30 \end{aligned}$$

Définition 3 Variable libre

Dépendamment de la portée d'une expression, une variable peut être lié *bound* à une abstraction si la variable de l'abstraction correspond à cette variable. On dit qu'un variable est libre lorsqu'elle ne correspond pas à la variable ID de l'abstraction mais apparaît quand même dans l'expression de cette abstraction.

$$\lambda x\ .\ (+1)\ y$$

Est une abstraction qui contient la **variable libre** y puisque l'abstraction est exprimé en terme de y, alors que λ est suivit de la métavariable x.

Exemple 7 Identification de variable libre

$$\lambda x\ .\ xyz : \textbf{pour } x\ ?$$

Non, puisque x apparaît dans l'expression de l'abstraction et que l'abstraction contient de métavariable x ; x est lié à la fonction.

$$\lambda x\ .\ xyz : \textbf{pour } y\ ?$$

Oui, y n'est pas la métavariable de l'abstraction.

$$(\lambda x\ .\ (+\ x\ 1))\ x : \textbf{pour } x\ ?$$

Le premier x est à l'intérieur d'une abstraction qui a une métavariable du même nom. Le second x est un paramètre formel utilisé pour évaluer l'expression de l'abstraction et est donc une variable libre.

$$\lambda x\ .\ \lambda y\ .\ \lambda z\ .\ zyx : \textbf{pour } z\ ?$$

Non, puisque z apparaît dans l'expression et a la même identité que la métavariable de l'abstraction.

$$(\lambda x\ .\ z\ \texttt{foo})(\lambda y\ .\ yx) : \textbf{pour } x\ ?$$

Oui, puisque x n'apparaît pas dans le corps de la première expression, malgré l'existance de la métavariable x. Et pour la seconde expression, la métavariable est y ; le x qui apparaît dans le corps de cette expression n'est donc pas lié à la métavariable de l'expression.

Règles d'expression libre

La variable x est **libre** dans une expression E **Si** :

$$E = x$$
$$E = \lambda y\ .\ E_1, \text{où } y! = x \text{ et } x \text{ libre dans } E_1$$
$$E = E_1E_2, \text{où } x \text{est libre dans } E_1$$
$$E = E_1E_2, \text{où } x \text{est libre dans } E_2$$

La première règle dit que si l'expression est simplement la variable x, x est libre dans cette expression. La seconde règle dit que si x est libre dans une expression E si x est libre dans l'abstraction interne E_1, tant que la métavariable de l'abstraction externe n'est pas équivalente à x Les deux dernières règle indiquent que x est libre dans une expression E composée de deux expression E_1 E_2, tant que x est libre dans une de ces deux expressions.

Exemple 8 Variable libre

Est-ce que x est libre dans les expression suivantes :

$$x\lambda x\ .\ x$$

Oui, puisque l'expression désambiguié est $x(\lambda x\ .\ x)$ et obéit à la syntaxe $E \to E_1E_2$ où x de E_1 est tel que $E_1 = x$ et est donc une variable libre. Ainsi l'expression globale $E = E_1E_2$ est libre puisque x est libre dans E_1.

$$\lambda x\ .\ yx$$

Non puisque x fait parti de corps de l'abstraction ; la métavariable de l'abstraction est aussi x, donc x n'est pas libre dans cette expression.

Définition 4 Combinateur

Une expression est un combinateur si elle ne contient aucune variable libre.

Exemple 9

$$\lambda x\ .\lambda y\ .\ xyx$$

Est un combinateur, puisque y est lié à l'abstraction interne et x est lié à l'abstraction externe

$$\lambda x\ .\ x$$

Est un combinateur puisque x est lié à l'abstraction dont la métavariable a la même idendité

$$\lambda z\ .\ \lambda x\ .\ xyz$$

N'est pas un combinateur, puisque y n'est pas lié

Définition 5 Variable liée

Une variable liée est une variable qui n'est pas libre. Lorsqu'une variable est liée, il important de déterminer à quelle abstraction elle est liée.

Note :

Les variables **libres** et **liées** en programmation fonctionnelle se comportent comme les variable globales et locales dans les langages impératifs. Lorsqu'un variable est libre dans une fonction interne il est possible qu'elle soit tout de même lié à une abstraction externe.

Exemple 10

L'expression suivante s'évalue comme suit

$$(\lambda x\ .\ x(\lambda x\ .\ x))1 = 1(\lambda x\ .\ x)$$

Seul le premier x de l'expression a été substitué pour l'argument 1, puisque seul ce x est lié à l'abstraction externe. Le second x est lié à l'abstraction interne.

On peut aussi considérer le corps de l'abstraction externe :

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

Dans cette expression, $(E \to E_1E_2)$, on a $E_1 = x$ et donc le premier x est une variable libre. Sachant que le premier x est une variable libre, on peut déduire que ce x est lié à une de niveau supérieur (abstraction externe), pourvue que cette abstraction contient la métavariable x.

Equivalence

Définition 6 α-equivalence

Deux fonctions sont alpha équivalentes lorsqu'elles diffèrent uniquement pas le nom des **variables liées**

$$E_1 =_\alpha E_2$$

Opérations de renommage

On utilise la syntaxe $E\{y\ /\ x\}$ pour signifier qu'on **substitue** toutes les instances de x par y dans l'expression E. La substitution doit obéir au règles suivantes

$E\{y\ /\ x\}$ **:**

$$x\{y\ /\ x\} = y$$
$$z\{y\ /\ x\} = z \textbf{ si } z \neq x$$
$$(E_1E_2)\{y\ /\ x\} = (E_1\{y\ /\ x\})(E_2\{y\ /\ x\})$$
$$\lambda x\ .\ E\{y\ /\ x\} = \lambda y\ .\ E\{y\ /\ x\}$$
$$\lambda z\ .\ E\{y\ /\ x\} = \lambda z\ .\ E\{y\ /\ x\} \textbf{ si } z \neq x$$

Exemple 11 Substitutions

$$\begin{aligned}\lambda x\ .\ x\{\texttt{foo}\ /\ x\} &= \lambda\texttt{foo}\ .\ x\ \{\texttt{foo}\ /\ x\}\\ &= \lambda\texttt{foo}\ .\ \texttt{foo}\end{aligned}$$

$$\begin{aligned}&(\lambda x\ .\ x(\lambda y\ .\ xyzy)xy)\ \{\texttt{bar}\ /\ x\}\\ &= \lambda\texttt{bar}\ .\ \Big(x(\lambda y\ .\ xyzy)xy\Big)\{\texttt{bar}\ /\ x\}\\ &= \lambda\texttt{bar}\ .\ \texttt{bar}\Big(\lambda y\ .\ xyzy\Big)xy\ \ \{\texttt{bar}\ /\ x\}\\ &= \lambda\texttt{bar}\ .\ \texttt{bar}\Big(\lambda y\ .\ xyzy\{\texttt{bar}\ /\ x\}\Big)\Big(xy\ \{\texttt{bar}\ /\ x\}\Big)\\ &= \lambda\texttt{bar}\ .\ \texttt{bar}\Big(\lambda y\ .\ \texttt{bar}yzy\Big)\Big(\texttt{bar}y\Big)\end{aligned}$$

Théorème 1 Expressions α-équivalentes

Tant que la varable y n'appartient pas à l'expressions E, toutes les expression $\lambda y\ .\ E\{y\ /\ x\}$ sont alpha équivalentes à l'expression $\lambda x\ .\ E$

$$\lambda x\ .\ E =_\alpha \lambda y\ .\ E\{y\ /\ x\}\Big|\ y \notin E$$

Opération de substitution

Le **renommage** permet uniquement de remplacer une variable par une autre ($\{x\ /\ y\}$). Or, pour réduire une expression, il faut effectuer une **substitution**

$$E[x \to N]\ ;\ \Big|\ E = E_1, N = E_2 \text{ sont des } \lambda\text{-exp.}$$

Cette expression revient à dire qu'on remplace x avec N où E et N sont des λ-expressions et x est une **variable non liée** .

Exemple 12

$(+\ x\ 1)[x \to 2] = (+\ 2\ 1)$
$(\lambda x\ .\ +\ x\ 1)[x \to 2] = (\lambda x\ .\ +\ x\ 1)$
$(\lambda x\ .\ y\ x)[y \to \lambda z\ .\ xz]$
$=$
$(\lambda x\ .(\lambda z\ .\ xz)x)$ **Intermédiare**
$=$
$(\lambda w\ .(\lambda z\ .\ xz)w)$

Règles de substitution

$x[x \to N] = N$
$y[x \to N] = y$ **si** $x \neq y$
$(E_1\ E_2)[x \to N] = \big(E_1\ [x \to N]\big)\big(E_2\ [x \to N]\big)$
$\lambda x\ .\ E[x \to N] = \lambda x\ .\ E$
$\lambda y\ .\ E[x \to N] = \lambda y\ .\ \big(E[x \to N]\big)$
où $x \neq y$, et y n'est pas libre dans N
$\lambda y\ .\ \big(E[x \to N]\big) = \big(\lambda y'\ .\ E\{y'\ /\ y\}[x \to N]\big)$
où $x \neq y, y$ libre dans N, et
y' est un nouveau nom

Exemple 13

$(\lambda x\ .\ x)[x \to \texttt{foo}] = (\lambda x\ .\ x)$
Obéit à $\lambda x\ .\ E[x \to N] = \lambda x\ .\ E$

$(+\ 1\ x)[x \to 2]$
$= \big(((+\ 1)\ x)[x \to 2]\big)\big(x[x \to 2]\big)$
Obtéit à$(E_1\ E_2)[x \to N]$
$= (E_1[x \to N])\ (E_2[x \to N])$
$\leftrightarrow \big(+[x \to 2]\big)\big(1[x \to 2]\big)\big(x[x \to 2]\big)$
$= +(\ 1\ 2)$

$(\lambda x\ .\ yx)[y \to \lambda z\ .\ xz]$
$= \lambda w\ .\big(\ yx\big)\{w\ /\ x\}[y \to \lambda z\ .\ xz]$
$= \lambda w\ .\ \big(yw\big)[y \to \lambda z\ .\ xz]$
$= \lambda w\ .\ \big(y[y \to \lambda z\ .\ xz]\big)\big(w[y \to \lambda z\ .\ xz]\big)$
$= \lambda y\ .\ (\lambda z\ .\ xz)(w)$

Execution

Définition 7 Exécution

Engendre une séquence de termes qui est le résultat d'appels et d'invocations de fonctions

▷ Chaque étape du processus d'**exécution** est appelé β-reduction.

▷ On peut seulemnet β-réduire des β-redux ; des expressions sous la forme *application*

- $(\lambda x\ .\ E)\ N$

▷ Une β-réduction est définit par :

$(\lambda x\ .\ E)\ N$ se β-réduit à $E[x \to N]$

Exemple 14 Exécutions

Exercice 1 :

$$(\lambda x\ .\ x)y = x[x \to y] = y$$

Exercice 2 :

$$\begin{aligned}
&\big(\lambda x\ .\ x(\lambda x\ .\ x)\big)ur \\
&= \Big(\big(\lambda x\ .\ x(\lambda x\ .\ x)\big)u\Big)r \\
&= \Big(x(\lambda x\ .\ x)[x \to u]\Big)r \\
&= \Big(\big(x[x \to u]\big)\big(\lambda x\ .\ x)[x \to u]\big)\Big)r \\
&= \Big(u\ \lambda x\ .\ x\Big)r
\end{aligned}$$

Exercice 3 :

$$\begin{aligned}
&\big(\lambda x\ .\ x(\lambda x\ .\ x)\big)(ur) \\
&= x(\lambda x\ .\ x)[x \to (ur)] \\
&= \big(x[x \to (ur)]\big)\big(\lambda x\ .\ x[x \to (ur)]\big) \\
&= (ur)(\lambda x\ .\ x)
\end{aligned}$$

Logique booléenne

▷ `True` est une *fonction* qui prend deux argument et retourne le **premier**

$$T = \lambda x\ .\ \lambda y\ .\ x$$

▷ `False` est une *fonction* qui prend deux argument et retourne le **second**

$$F = \lambda x\ .\ \lambda y\ .\ y$$

Exemple 15

$$\begin{aligned}
(\lambda x\ .\ \lambda y\ .\ x)\ a\ b &= ((\lambda x\ .\ \lambda y\ .\ x)\ a)\ b \\
&= \big((\lambda y\ .\ x)[x \to a]\big)b \\
&= \big(\lambda y\ .\ a\big)b \\
&= \big(\lambda y\ .\ a\big)[y \to b] \\
&= a
\end{aligned}$$

▷ `AND` est une *fonction* qui prend deux arguments et renvoit `True` lorsqu'ils sont tous deux `True` et `False` autrement.

$$\texttt{AND} = \lambda x\ .\ \lambda y\ .\ xyx$$

Cas 1 : $x = T$, $y = T$

$$\begin{aligned}
(\lambda x.\lambda y.xyx)\ T\ T &= (\lambda y.T\ y\ T)\ T \\
&= T\ T\ T \\
&= T
\end{aligned}$$

Cas 2 : $x = T$, $y = F$

$$\begin{aligned}
(\lambda x.\lambda y.xyx)\ T\ F &= (\lambda y.T\ y\ T)\ F \\
&= T\ F\ T \\
&= F
\end{aligned}$$

Cas 3 : $x = F$, $y = T$

$$\begin{aligned}
(\lambda x.\lambda y.xyx)\ F\ T &= (\lambda y.F\ y\ F)\ T \\
&= F\ T\ F \\
&= F
\end{aligned}$$

Cas 4 : $x = F$, $y = F$

$$\begin{aligned}
(\lambda x.\lambda y.xyx)\ F\ F &= (\lambda y.F\ y\ F)\ F \\
&= F\ F\ F \\
&= F
\end{aligned}$$

▷ `NOT` est une *fonction* qui prend un argument et renvoit l'opposé de celui-ci

Note :

Puisque l'argument de `NOT` est une fonction `True` ou `False`, en principe, `NOT` doit prendre deux arguments. La fonction que prend `NOT` (l'argument de `NOT`) renverra le premier argument si cette fonction est `True` La fonction que prend `NOT` renverra le second argument si cette fonction est `False`. C'est pourquoi dans le corps de l'expression `NOT` il y a deux fonctions, soit `False` et `True` placées **devant** la fonction qui sera substituée par x ; ces fonctions servent d'argument à la fonctione substuée par x, qui se trouve à être l'argument de `NOT`.

$$\texttt{NOT} = \lambda x\ .\ xFT$$

Cas 1 : $x = T$

$$\begin{aligned}
(\lambda x.xF\ T)T &= x\ F\ T[x \to T] \\
&= TFT \\
&= (\lambda x.\lambda y.x)FT \\
&= (\lambda y.x)[x \to F]T \\
&= (\lambda y.F)T \\
&= (F[y \to T]) \\
&= F
\end{aligned}$$

Cas 2 : $x = F$

$$\begin{aligned}
(\lambda x.xF\ T)T &= x\ F\ T[x \to F] \\
&= FFT \\
&= (\lambda x.\lambda y.y)FT \\
&= (\lambda y.y)[x \to F]T \\
&= (\lambda y.y)T \\
&= (y[y \to T]) \\
&= T
\end{aligned}$$

Branche `if` Les branches permettent de changer le flow d'exécution d'une programme, en fonction de la condition du branchement et de la variable d'entrée.

L'architecture d'une *fonction* `if`, en lambda calcul, devrait donc être telle que la fonction prend un booléen, comporte une branche `True` et comporte une seconde branche `False`.

```
if c then
  a
else
  b
```

Le squelette de la fonction serait :

$$\texttt{if}\ c\, a\, b$$

La fonction `if` devrait retourner a ou b, dépendamment du booléen présenté en argument. La fonction `tx` prend alors trois paramètres. Si le paramètre booléen est vrai, elle retourne le paramètre a, qui se trouve à être le premier paramètre après le paramètre booléens. Si le paramètre booléen est faux, elle retourne le second paramètre après le paramètre booléen, soit b.

On constate que le paramètre c est simplement la fonction `True` ou la fonction `False`. On a :

$$\begin{aligned}\texttt{if}\ T\, a\, b &= a\\ \texttt{if}\ F\, a\, b &= b\end{aligned}$$

Ainsi, la fonction `if` semble avoir la même sémantique que la fonction identité $\lambda x\,.\,x$ qui retourne la valeur d'entrée.

Ainsi, on a

$$\begin{aligned}&\texttt{if}\ T\ a\ b \leftrightarrow (\lambda w\,.\,w)\lambda x\,.\,\lambda y\,.\,x\ a\ b\\ &\textbf{et}\\ &\texttt{if}\ F\ a\ b \leftrightarrow (\lambda w\,.\,w)\lambda x\,.\,\lambda y\,.\,y\ a\ b\end{aligned}$$

Church's nurmerals L'intuition derrière les nombre de Church est qu'un nombre correspond au nombre de fois où une fonction quelconque f passé en argument sur la fonction du nombre est appliqué sur la variable x

$$\begin{aligned}0 &= \lambda f.\lambda x.x\\ 1 &= \lambda f.\lambda x.f\,x\\ 2 &= \lambda f.\lambda x.f\,(f\,x)\\ 3 &= \lambda f.\lambda x.f\,(f\,(f\,x))\\ 4 &= \lambda f.\lambda x.f\,(f\,(f\,(f\,x)))\end{aligned}$$

Exemple 16

$$\begin{aligned}4\,a\,b &= \Big(\lambda f.\lambda x.f(f(f(fx)))a\Big)b\\ &= \Big(\lambda x.f(f(f(fx)))a\Big)[f \to a]\ b\\ &= \Big(\lambda x.a(a(a(ax)))\Big)\ b\\ &= \Big(a(a(a(ax)))\Big)[x \to b]\ b\\ &= \Big(a(a(a(ab)))\Big)\end{aligned}$$

Fonction successeur

$$\texttt{succ} = \lambda n.\lambda f.\lambda x.f(n\,f\,x)$$

La fonction successeur en lambda calcul, notée `succ`, est utilisée pour ajouter 1 à un nombre de Church. En d'autres mots, La fonction prend un nombre de Church n comme entrée et renvoie en sortie un nouveau nombre de Church qui représente $n+1$.

Exemple 17 Successeur de 1

$$\begin{aligned}&\Big(\lambda n.\lambda f.\lambda x.f(n\,f\,x)\Big)1\\ &= \Big(\lambda n.\lambda f.\lambda x.f(n\,f\,x)\Big)(\lambda f.\lambda x.f\,x)\\ &= \lambda f.\lambda x.f\Big((\lambda f.\lambda x.f\,x)\,f\,x\Big)\\ &= \lambda f.\lambda x.f\Big((\lambda x.f\,x)[f \to f]\,x\Big)\\ &= \lambda f.\lambda x.f\Big(f\,x\Big)\\ &= \lambda f.\lambda x.f(f\,x)\\ &= \lambda f.\lambda x.f(f\,x)\end{aligned}$$

Addition La fonction d'addition prend deux paramètre et retourne une fonction qui correspond au nombre de Church représentant la somme des deux paramètres

Cela peut être interprété comme appliquer f n fois à x, puis appliquer ce résultat f m fois à ce résultat.

$$\texttt{add} = \lambda m.\lambda n.\lambda f.\lambda x.n\,f\ (m\,x\,f)$$

Exemple 18 Addition de 0 et 1

$$\begin{aligned}&(\lambda n.\lambda m.\lambda f.\lambda x.nf(mfx))\ 0\,1\\ &= (\lambda m.\lambda f.\lambda x.0f(mfx))\ 1\\ &= \lambda f.\lambda x.0f(1fx)\\ &= \lambda f.\lambda x.(\lambda f.\lambda x.x)f(fx)\\ &= \lambda f.\lambda x.(\lambda x.x)[x \to fx]\\ &= \lambda f.\lambda x.x\\ &= \lambda f.\lambda x.fx\end{aligned}$$

Multiplication

$$\lambda n.\lambda m.m(addn)0$$

Exemple 19 Multiplication de 0 et 1

$$\begin{aligned}&(\lambda n.\lambda m.m(\mathrm{add}\,n)\,0)\,0\,1\\ &= (\lambda m.m(\mathrm{add}\,0)\,0)\,1\\ &= 1(\mathrm{add}\,0)\,0\\ &= (\lambda f.\lambda x.fx)(\mathrm{add}\,0)\,0\\ &= (\lambda f.\lambda x.fx)(\lambda m.\lambda f.\lambda x.0f(mfx))\,0\\ &= (\lambda f.\lambda x.fx)(\lambda f.\lambda x.0f(0fx))\\ &= (\lambda f.\lambda x.fx)(\lambda f.\lambda x.x)\\ &= \lambda x.(\lambda f.\lambda x.x)x\\ &= \lambda x.x\\ &= \lambda f.\lambda x.x\end{aligned}$$

Y-combinator Le combinateur Y est une fonction qui, à cause de sa structure particulière parvient à s'autorépliquer lorsqu'un argument lui est appliqué. La fonction résultante l'application Ya est YYa, qui à son tour, engendre $YYYa$, puis $YYYYa$, ainsi de suite.

$$Y = \Big(\lambda y.\lambda x.y(x\,x\,y)\Big)\Big(\lambda y.\lambda x.y(x\,x\,y)\Big)$$

Exemple 20

$$\begin{aligned}&Y\mathrm{foo} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))\\ &\mathrm{foo}\\ &= (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))\mathrm{foo}\\ &= (\lambda x.\mathrm{foo}(xx))(\lambda x.\mathrm{foo}(xx))\\ &= \mathrm{foo}((\lambda x.\mathrm{foo}(xx))(\lambda x.\mathrm{foo}(xx)))\\ &= \mathrm{foo}(Y\mathrm{foo})\\ &= \mathrm{foo}(\mathrm{foo}(Y\mathrm{foo}))\\ &= \mathrm{foo}(\mathrm{foo}(\mathrm{foo}(Y\mathrm{foo})))\\ &\vdots\end{aligned}$$