Programmation fonctionnelle

Simon Robillard

13 octobre 2023

Section 1

Syllabus

Objectifs du cours

- comprendre les mécanismes propres aux langages de programmation fonctionnels (et au lambda-calcul)
- utiliser les aspects fonctionnels du langage Scala

Fonctionnement du cours : classe inversée

un seul CM (aujourd'hui) + 5 séances de TD/TP

- **chez vous**
  - des notes de cours à lire obligatoirement
  - des liens vers des ressources additionnelles à consulter facultativement
- **en classe**
  - une session de questions/réponses pour clarifier les notes de cours
  - un QCM sur les dernières notes de cours
  - TD et/ou TP

Où trouver le cours ?

- les notes de cours et les fiches de TD seront mises en ligne chaque semaine sur Moodle
- le code des TP sera disponible en ligne sur Gitlab
- contact mail : `simon.robillard@umontpellier.fr`

Évaluation

La note de programmation fonctionnelle compte pour 30% de la ressource R5.A.05

- notes de QCM (15%)
- un DS (15%)

Section 2

Introduction

Qu'est-ce qu'un langage fonctionnel ?

Les fonctions comme citoyens de première classe

- les fonctions sont des valeurs !
- une fonction peut prendre pour argument une fonction
- une fonction peut retourner une fonction
- il est possible de déclarer simplement une fonction sans lui attribuer un nom

Qu'est-ce qu'un langage fonctionnel ?

Les fonctions comme citoyens de première classe

- les fonctions sont des valeurs !
- une fonction peut prendre pour argument une fonction
- une fonction peut retourner une fonction
- il est possible de déclarer simplement une fonction sans lui attribuer un nom

Qu'est-ce qu'une fonction ?

"boîte noire" qui prend une (ou des) valeur(s) en entrée et produit une valeur en sortie

- ne fait rien d'autre, en particulier ne modifie pas l'état d'un quelconque système
- cette définition est ce qu'on appelle une fonction *pure*, proche de la définition utilisée en mathématiques

À quoi servent les langages fonctionnels ?

- à faire de la programmation haut-niveau
  - pas besoin de se préoccuper de la gestion de la mémoire
  - les structures de données ressemblent à leur description abstraite
- à composer et réutiliser du code
  - nouvelles façons de combiner des fonctions
  - un système de typage statique fort nous permet de s'assurer que ces combinaisons sont correctes

Même si les langages 100% fonctionnels restent rares, on trouve désormais des éléments fonctionnels dans la plupart des langages courants (Java, C++, Python, et même récemment Excel !)

- **SCA**lable **LA**nguage
- développé en Suisse (EPFL) depuis 2001
- plusieurs paradigmes
  - programmation fonctionnelle
  - programmation impérative
  - programmation orientée objet
- tourne sur la JVM (Java Virtual Machine)
  - facile de combiner Java et Scala

Scala et les entreprises

Scala est utilisé par

- LinkedIn
- Twitter
- Netflix
- Tumblr
- Foursquare
- AirBnB
- ...

pour programmer des applications

- backend web
- big data
- distribuées

Les frameworks les plus communs

- Apache Spark (calcul distribué)
- Akka (application concurrentes)
- Apache Kafka (*event streaming*)

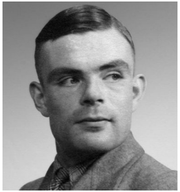
`https://sysgears.com/articles/how-tech-giants-use-scala`

Section 3

Le lambda-calcul

Un peu d'histoire

Dans les années 1930, on se pose la question "qu'est-ce que calculer ?"



Alan Turing (1912-1954)

- notion d'une "machine" mathématique qui lit et écrit des données sur un ruban, d'après des instructions
- inspirera les langages impératifs

Alonzo Church (1903-1995)

- formalise la notion de fonction calculable via le *lambda-calcul*
- inspirera les langages fonctionnels

Comment décrire des fonctions ?

- évaluer des expressions est une opération naturelle
- exemple : évaluez l'expression

$$5 \times 7 \geq 36 + 0 \times (4 + 2)$$

Comment décrire des fonctions ?

- évaluer des expressions est une opération naturelle
- exemple : évaluez l'expression

$$5 \times 7 \geq 36 + 0 \times (4 + 2)$$

- avec le lambda-calcul, Church montre que la notion d'évaluation est très puissante : toute fonction calculable peut se décrire comme une expression
- exécuter un programme fonctionnel = évaluer une expression

Le lambda-calcul : syntaxe abstraite

Soit $X = x, y, z, \ldots$ un ensemble (infini) de *variables*
Les λ-termes sont formés de la manière suivante

1. si x est une variable, alors

   $$\boldsymbol{x} \text{ est un } \lambda\text{-terme}$$

2. si x est une variable et t un λ-terme, alors

   $$\boldsymbol{\lambda x.t} \text{ est un } \lambda\text{-terme}$$

   qui désigne une fonction anonyme (lambda-abstraction) qui à x associe t

3. si t_1 et t_2 sont des λ-termes, alors

   $$\boldsymbol{t_1}\ \boldsymbol{t_2} \text{ est un } \lambda\text{-terme}$$

   qui désigne l'application de la fonction t_1 à l'argument t_2

Quelques exemples

$\lambda x.x$	$=$	fonction qui retourne son argument inchangé (fonction identité)
$\lambda x.y$	$=$	fonction qui retourne y, quel que soit son argument
$\lambda x.y\, x$	$=$	fonction qui prend un argument x et lui applique une fonction y
$\lambda x.\lambda y.y$	$=$	fonction qui retourne la fonction identité
$(\lambda x.x)(\lambda x.\lambda y.y)$	$=$	la fonction identité appliquée à un argument

Notes sur la syntaxe concrète : parenthèses

par défault, l'application associe à gauche

- $x\,y\,z$ est donc équivalent à $(x\,y)\,z$
- si on veut écrire $x\,(y\,z)$ les parenthèses sont obligatoires

les parenthèses peuvent aussi être nécessaires pour délimiter une lambda-abstraction

- $(\lambda x.x)\,y$ signifie $\lambda x.x$ appliqué à y
- $\lambda x.x\,y$ signifie une fonction qui prend un argument x et renvoie x appliqué à y

Évaluer une lambda expression

Une seule règle :

$$(\lambda x.t_1)\, t_2 \text{ se réécrit en } t_1[x \mapsto t_2]$$

c'est-à-dire l'expression t_1 où chaque occurrence (libre) de x est remplacée par t_2

Exemples

1. $(\lambda x.x)\, y$ se réduit à y
2. $(\lambda x.z)\, y$ se réduit à z

- cette transformation est appelée β-*réduction*
- quand cette règle n'est plus applicable, le terme est dit en *forme normale*

Exercice 1 : évaluation

Utilisons les notations suivantes

- $t \equiv \lambda x_0.\lambda y_0.x_0$
- $f \equiv \lambda x_1.\lambda y_1.y_1$
- $n \equiv \lambda x_2.\lambda y_2.\lambda z_2.x_2\ z_2\ y_2$
- $ite \equiv \lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3$

Évaluer

1. $n\ t$
2. $n\ f$
3. $ite\ t\ a\ b$
4. $ite\ f\ a\ b$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$\begin{aligned} & n\,t \\ \equiv\quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\,(\lambda x_0.\lambda y_0.x_0) \end{aligned}$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$\begin{aligned} & n\, t \\ \equiv\;\; & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\,(\lambda x_0.\lambda y_0.x_0) \end{aligned}$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$\begin{array}{rl} & n\; t \\ \equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\,(\lambda x_0.\lambda y_0.x_0) \\ \rightarrow_\beta & \lambda y_2.\lambda z_2.(\lambda x_0.\lambda y_0.x_0)\; z_2\; y_2 \end{array}$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$
\begin{array}{rl}
& n\, t \\
\equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\, (\lambda x_0.\lambda y_0.x_0) \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda x_0.\lambda y_0.x_0)\, z_2\, y_2
\end{array}
$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$
\begin{array}{rl}
 & n\ t \\
\equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\ z_2\ y_2)\ (\lambda x_0.\lambda y_0.x_0) \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda x_0.\lambda y_0.x_0)\ z_2\ y_2 \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda y_0.z_2)\ y_2
\end{array}
$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$
\begin{array}{rl}
 & n\ t \\
\equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\ z_2\ y_2)\ (\lambda x_0.\lambda y_0.x_0) \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda x_0.\lambda y_0.x_0)\ z_2\ y_2 \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda y_0.z_2)\ y_2
\end{array}
$$

Exercice 1.1 : correction

À chaque étape, identifier une λ-abstraction appliquée à un terme (ce qu'on appelle une *rédex*) et la réduire.

$$
\begin{aligned}
& n\,t \\
\equiv\quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\,(\lambda x_0.\lambda y_0.x_0) \\
\to_\beta\quad & \lambda y_2.\lambda z_2.(\lambda x_0.\lambda y_0.x_0)\, z_2\, y_2 \\
\to_\beta\quad & \lambda y_2.\lambda z_2.(\lambda y_0.z_2)\, y_2 \\
\to_\beta\quad & \lambda y_2.\lambda z_2.z_2
\end{aligned}
$$

Exercice 1.2 : correction

$$
\begin{array}{rl}
 & n\,f \\
\equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\,(\lambda x_1.\lambda y_1.y_1)
\end{array}
$$

Exercice 1.2 : correction

$$
\begin{aligned}
& n\, f \\
\equiv \quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\, (\lambda x_1.\lambda y_1.y_1)
\end{aligned}
$$

Exercice 1.2 : correction

$$
\begin{aligned}
& n\, f \\
\equiv\quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\, (\lambda x_1.\lambda y_1.y_1) \\
\to_\beta\quad & \lambda y_2.\lambda z_2.(\lambda x_1.\lambda y_1.y_1)\; z_2\; y_2
\end{aligned}
$$

Exercice 1.2 : correction

$$
\begin{aligned}
& n\,f \\
\equiv\quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\,(\lambda x_1.\lambda y_1.y_1) \\
\rightarrow_\beta\quad & \lambda y_2.\lambda z_2.(\lambda x_1.\lambda y_1.y_1)\; z_2\; y_2
\end{aligned}
$$

Exercice 1.2 : correction

$$
\begin{aligned}
& n\, f \\
\equiv \quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\, z_2\, y_2)\, (\lambda x_1.\lambda y_1.y_1) \\
\to_\beta \quad & \lambda y_2.\lambda z_2.(\lambda x_1.\lambda y_1.y_1)\, z_2\, y_2 \\
\to_\beta \quad & \lambda y_2.\lambda z_2.(\lambda y_1.y_1)\, y_2
\end{aligned}
$$

Exercice 1.2 : correction

$$
\begin{array}{rl}
 & n\, f \\
\equiv & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\,(\lambda x_1.\lambda y_1.y_1) \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda x_1.\lambda y_1.y_1)\; z_2\; y_2 \\
\to_\beta & \lambda y_2.\lambda z_2.(\lambda y_1.y_1)\; y_2
\end{array}
$$

Exercice 1.2 : correction

$$
\begin{aligned}
& n\, f \\
\equiv\quad & (\lambda x_2.\lambda y_2.\lambda z_2.x_2\; z_2\; y_2)\,(\lambda x_1.\lambda y_1.y_1) \\
\rightarrow_\beta\quad & \lambda y_2.\lambda z_2.(\lambda x_1.\lambda y_1.y_1)\; z_2\; y_2 \\
\rightarrow_\beta\quad & \lambda y_2.\lambda z_2.(\lambda y_1.y_1)\; y_2 \\
\rightarrow_\beta\quad & \lambda y_2.\lambda z_2.y_2
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{array}{rl}
& ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\,(\lambda x_0.\lambda y_0.x_0)\ a\ b
\end{array}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\ & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{array}{rl}
& ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b
\end{array}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv \quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\to_\beta \quad & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\to_\beta \quad & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{array}{rl}
 & ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b
\end{array}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\rightarrow_\beta\quad & (\lambda x_0.\lambda y_0.x_0)\ a\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\rightarrow_\beta\quad & (\lambda x_0.\lambda y_0.x_0)\ a\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{array}{rl}
 & ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\to_\beta & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\to_\beta & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\to_\beta & (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\to_\beta & (\lambda y_0.a)\ b
\end{array}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\to_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\to_\beta\quad & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\to_\beta\quad & (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\to_\beta\quad & (\lambda y_0.a)\ b
\end{aligned}
$$

Exercice 1.3 : correction

$$
\begin{array}{rl}
 & ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\rightarrow_\beta & (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta & (\lambda y_0.a)\ b \\
\rightarrow_\beta & a
\end{array}
$$

Exercice 1.3 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\ & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\ & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\ & (\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ a\ z_3)\ b \\
\rightarrow_\beta\ & (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\ & (\lambda y_0.a)\ b \\
\rightarrow_\beta\ & a
\end{aligned}
$$

Quand il y a plusieurs rédex, plusieurs façons d'évaluer sont possibles, par ex. :

$$
\begin{aligned}
& (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_0.\lambda y_0.x_0)\ a\ b \\
\rightarrow_\beta\ & (\lambda y_3.\lambda z_3.(\lambda x_0.\lambda y_0.x_0)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\ & (\lambda y_3.\lambda z_3.(\lambda y_0.y_3)\ z_3)\ a\ b \\
\rightarrow_\beta\ & (\lambda y_3.\lambda z_3.y_3)\ a\ b \\
\rightarrow_\beta\ & (\lambda z_3.a)\ b \\
\rightarrow_\beta\ & a
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\,(\lambda x_1.\lambda y_1.y_1)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\ & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\,(\lambda x_1.\lambda y_1.y_1)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\to_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{array}{rl}
 & ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b
\end{array}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& ite\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b \\
\rightarrow_\beta\quad & (\lambda x_1.\lambda y_1.y_1)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b \\
\rightarrow_\beta\quad & (\lambda x_1.\lambda y_1.y_1)\ a\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{array}{rl}
 & ite\ t\ a\ b \\
\equiv & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b \\
\rightarrow_\beta & (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta & (\lambda y_1.y_1)\ b
\end{array}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\rightarrow_\beta\quad & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b \\
\rightarrow_\beta\quad & (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\rightarrow_\beta\quad & (\lambda y_1.y_1)\ b
\end{aligned}
$$

Exercice 1.4 : correction

$$
\begin{aligned}
& \mathit{ite}\ t\ a\ b \\
\equiv\quad & (\lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3)\ (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\to_\beta\quad & (\lambda y_3.\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ y_3\ z_3)\ a\ b \\
\to_\beta\quad & (\lambda z_3.(\lambda x_1.\lambda y_1.y_1)\ a\ z_3)\ b \\
\to_\beta\quad & (\lambda x_1.\lambda y_1.y_1)\ a\ b \\
\to_\beta\quad & (\lambda y_1.y_1)\ b \\
\to_\beta\quad & b
\end{aligned}
$$

Variables libres ?

Comment s'évalue $(\lambda x.\lambda x.x)\, a\, b$?

1. a
2. b

Variables libres ?

Comment s'évalue $(\lambda x.\lambda x.x)\, a\, b$?

1. $\cancel{a}$
2. b ✓

- on doit commencer par réécrire les occurrences *libres* de x
- le terme $\lambda x.x$ n'a pas d'occurrence libre !
- la variable x est *liée* au lambda le plus à l'intérieur dans l'expression

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

- comme dans les autres langages de programmation : une variable locale masque les variables du même nom définie à un niveau plus global

Renommage de variables

- on peut changer le nom des variables (α-conversion)

  exemple : $\lambda x.\lambda x.x$ est équivalent à $\lambda x.\lambda x_0.x_0$
- parfois le renommage est obligatoire pour éviter la *capture de variable*

  exemple : comment s'évalue $(\lambda x.\lambda y.x\,y)\,y$?

  1. $\lambda y.y\,y$
  2. $\lambda y_0.y\,y_0$

Renommage de variables

- on peut changer le nom des variables (α-conversion)

  exemple : $\lambda x.\lambda x.x$ est équivalent à $\lambda x.\lambda x_0.x_0$

- parfois le renommage est obligatoire pour éviter la *capture de variable*

  exemple : comment s'évalue $(\lambda x.\lambda y.x\, y)\, y$?

  1. $\cancel{\lambda y.y\, y}$ cette réduction est incorrecte à cause d'une capture de la variable y
  2. $\lambda y_0.y\, y_0$ ✓

À retenir

- le partage des noms de variables peut compliquer l'évaluation
- il est utile de renommer la variable d'une λ-abstraction
  - si plusieurs λ-abstractions utilisent le même nom de variable (renommage facultatif, mais permet d'éviter des erreurs)
  - pour évaluer $(\lambda x.t_1)\, t_2$, si t_1 contient une variable liée de même nom qu'un des termes dans t_2 (renommage obligatoire pour éviter la capture)

Exercice 2 : codage des booléens

Rappel des notations et solutions de l'exercice 1

- $t \equiv \lambda x_0.y_0.x_0$
- $f \equiv \lambda x_1.y_1.y_1$
- $n \equiv \lambda x_2.\lambda y_2.\lambda z_2.x_2\ z_2\ y_2$
- $ite \equiv \lambda x_3.\lambda y_3.\lambda z_3.x_3\ y_3\ z_3$

expression	forme normale	α-équivalente à
$n\ t$	$\lambda y_2.\lambda z_2.z_2$	f
$n\ f$	$\lambda y_2.\lambda z_2.y_2$	t
$ite\ t\ a\ b$	a	a
$ite\ f\ a\ b$	b	b

Questions

1. comment utiliser ces termes pour encoder les booléens ?
2. comment encoder les opérateurs booléens AND, OR et XOR ?

L'expressivité du lambda-calcul

Données

- nous venons de voir qu'il est possible d'encoder les booléens
- il est possible d'encoder les entiers naturels (*entiers de Church*)

$$0 \equiv \lambda f.\lambda z.z \qquad 1 \equiv \lambda f.\lambda z.f\ z \qquad 2 \equiv \lambda f.\lambda z.f\ (f\ z) \qquad \ldots$$

- on peut *in fine* représenter toute donnée manipulable par un ordinateur

Algorithmes

- nous avons vu comment représenter les conditionnelles
- il est possible d'écrire des fonctions récursives (avec le *combinateur* Y par exemple)

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

La thèse Church-Turing

- Church et Turing ont tous deux voulu définir la notion de “calcul”
- ils ont créé deux modèles très différents l’un de l’autre
- ces deux modèles sont pourtant aussi expressifs l’un que l’autre
  - toute machine de Turing peut être traduite en un lambda-terme équivalent
  - et vice-versa
- on dit d’un modèle de calcul aussi expressif qu’il est Turing-complet

La thèse Church-Turing

- Church et Turing ont tous deux voulu définir la notion de “calcul”
- ils ont créé deux modèles très différents l'un de l'autre
- ces deux modèles sont pourtant aussi expressifs l'un que l'autre
  - toute machine de Turing peut être traduite en un lambda-terme équivalent
  - et vice-versa
- on dit d'un modèle de calcul aussi expressif qu'il est Turing-complet

La thèse* Church-Turing

Il n'existe pas de modèle de calcul plus expressif (que le lambda-calcul, ou les machines de Turing, ou autre modèle de calcul Turing-complet)

* il s'agit d'une affirmation non-démontrée (et probablement non-démontrable) mais très largement admise

Section 4

Scala

Et en Scala, ça donne quoi ?

- pour écrire une lambda-abstraction

```
(x: T) => e
```

- il faut préciser le type T de la variable x (par exemple `Int` ou `Boolean`)
- définir une fonction = donner un nom à une lambda-abstraction

```
def foo = (x: T) => e
```

- on peut aussi écrire

```
def bar(x: T) = e
```

Vers un langage de programmation

- les lambda-abstractions seules permettent de tout coder
- suffisant en théorie, mais pas très pratique !
- les langages de programmation fonctionnels (dont Scala) ont un langage plus riche
    - constantes booléennes, numériques, et autre types de données (que nous verrons plus tard)
    - fonctions prédéfinies ($+$, $\times$, $<$, *if* / *then* / *else*, ...)
    - règles d'évaluation habituelles pour ces fonctions
        - $1 + 1$ se réduit à 2
        - $3 < 1$ se réduit à *false*
        - *if false then* 5 *else* 4 se réduit à 4
        - ...

Demo avec le REPL Scala

Utiliser la commande `scala` pour lancer le REPL (*read-eval-print loop*)

Ressources supplémentaires

Vidéos avec sous-titres français disponibles

- *Lambda Calculus - Computerphile*
  Dans cette vidéo, Graham Hutton décrit les bases du lambda-calcul (similaire à ce cours).
  `https://youtu.be/eis11j_iGMs`
- *Essentials : Functional Programming's Y Combinator - Computerphile*
  Cette vidéo fait suite à la précédente et dévoile comment définir des fonctions récursives dans le lambda-calcul.
  `https://youtu.be/9T8A89jgeTI`