Programmation IV

.. a pour sujet :

les paradigmes de programmation avancés (et pas encore complètement standards)

En particulier :

- La Programmation Fonctionnelle
- La Programmation Logique

Les Paradigmes de la Programmation

Paradigme: “Un exemple qui sert de patron ou modèle”.

Dans ce cours: Un patron qui sert d'*école de pensées* pour programmer les ordinateurs.

Principaux paradigmes:

- La Programmation Impérative
- La Programmation Fonctionnelle
- La Programmation Logique
- Orthogonal à ce qui précède: l'Orientation-Objet.

La Programmation Impérative

On programme avec:

- Des variables (modifiables),
- Des affectations
- Des structures de contrôle, comme :

  If-then-else, loops, break, contine, return, goto.

Deux manières possibles de conceptualiser un programme :

- Comme une suite d'instructions pour une machine de Van Neuman
- Comme une relation entre prédicats.

Les Programmes comme Suites d'Instructions pour des Machines de Van Neuman

Variable	=	cellule mémoire
Référence à une variable	=	charger (load)
Affectation d'une variable	=	enregistrer (store)
Structures de contrôle	=	sauts (jumps)

Problème: Passage à l'échelle – nécessite de décrire les opérations mot à mot.

Les Programmes comme Transformeurs de Prédicats

- Un programme fait le lien entre une précondition et une postcondition.
- Les conditions sont des prédicats, c-à-d des formules booléennes portant sur des variables.
- Deux versions:
  - Le programme établit la postcondition après exécution, pourvu que la précondition soit remplie avant exécution (correction totale).
  - Le programme établit la postcondition après exécution, pourvu que la précondition soit remplie avant exécution et que le programme termine (correction partielle; on doit prouver la terminaison séparément).

Les Lois de la Programmation Impérative

$$\{P\}\ skip\ \{P\}$$

$$\{[e/x]Q\}\ x := e\ \{Q\}$$

$$\frac{\{P \wedge c\}\ s_1\{Q\} \qquad \{P \wedge \neg c\}s_2\ \{Q\}}{\{P\}\ \textbf{if}\,(c)\ s_1\ \textbf{else}\ s_2\ \{Q\}}$$

$$\frac{\{P \wedge c\ \}s\ \{P\}}{\{P\}\ \textbf{while}\,(c)\ s\ \{P \wedge \neg c\}}$$

$$\frac{P' \Rightarrow P \qquad \{P\}\ s\ \{Q\} \qquad Q \Rightarrow Q'}{\{P'\}\ s\ \{Q'\}}$$

(P est l'invariant de boucle dans la règle pour le *while*).

Exemple

Echanger deux variables:

$$\{y = Y, x = X\}$$
$$t := x$$
$$\{y = Y, t = X\}$$
$$x := y$$
$$\{x = Y, t = X\}$$
$$y := t$$
$$\{x = Y, y = X\}$$

Exemple: Recherche Linéaire

Si on définit l'invariant:

$$I \equiv i \leq a.length \wedge \forall j.0 \leq j < i.a(j) \neq x$$

Alors:

$\{\ true\ \}$
var $i := 0$
$\{\ I\ \}$
while $(i < a.length\ \&\&\ a(i)\ != x)$
$\{\ i < a.length \wedge a(i) \neq x \wedge \forall j.0 \leq j < i.a(j) \neq x\ \}$
$i = i + 1;$
$\{\ I\ \}$
$\{\ I \wedge \neg(i < a.length \wedge a(i) \neq x)\ \}$

Le dernier prédicat se simplifie en :

$$I \wedge i = a.length \vee a(i) = x.$$

Evaluation des Transformeurs de Prédicats

Avantage: Précision mathématique; peut être mécanisé.

Problème: Passage à l'échelle

Dans la version canonique des transformeurs de prédicats,

- Les programmes consistent en variables simples et tableaux
- Pas de structures de données définies par l'utilisateur.
- Pas d'encapsulation.

La programmation orientée-objet y peut-elle quelque chose ?

En fait, c'est tout le contraire.

La POO invalide les lois de la programmation impérative à cause des alias.

Exemple:

$$\{\ q.x = 1\ \}\ p.x := 2\ \{\ q.x = 1\ \}$$

Cela semble exact, comme conséquence de la loi pour l'affectation.

Mais cet argument est invalide si p and q sont des alias l'un de l'autre !

Donc, la loi pour l'affectation n'est plus valide si on permet les alias.

Par conséquent, les programmes qui mélangent changements d'état et alias sont très difficiles à comprendre et à vérifier.

Conséquences

La programmation impérative est contrainte par le principe de "Van Neumann".

On a besoin d'autres techniques pour définir des données de haut-niveau telles que les polynômes, les collections, les formes géométriques, les chaînes de caractères, les textes.

On a besoin de définir une théorie des polynômes, des collections, des textes, ...

Une théorie consiste en un ou plusieurs types de données avec des opérations qui sont liées entre elles par des lois.

Habituellement la théorie ne décrit pas de mutation.

Par ex.: La théorie des polynômes définit la somme de deux polynômes, par des lois telles que :

$$(a{*}x + b) \ + \ (c{*}x + d) \ = \ (a{+}c) * x + (b{+}d)$$

Elle ne définit pas d'opérateur pour modifier un coefficient dans un polynôme tout en gardant le même polynôme !

Autre ex.: La théorie des chaînes de caractères définit une opération de concaténation *concat* sur les chaînes de caractères, par des lois comme :

$$\text{"abc"}\ concat\ \text{"xyz"} \ = \ \text{"abcxyz"}$$

Elle ne définit pas un opérateur pour changer une lettre dans une chaîne de caractères tout en conservant l'identité de la chaîne !

Conséquences pour l'évolutivité des logiciels :

- On se concentre sur la définition de théorie pour de nouveaux opérateurs.
- On évite (diffère) les changements d'état.
- Les opérateurs sont des fonctions, qui sont souvent composées de fonctions plus simples.

La Programmation Fonctionnelle

Dans un sens restreint, la programmation fonctionnelle (PF) est la programmation sans les variables, les affectations ou les structures de contrôle impératives.

Dans un sens plus général, la programmation fonctionnelle c'est programmer en mettant l'accent sur les fonctions.

En particulier les fonctions deviennent des valeurs qui sont produites, consommées et composées par les programmes.

Tout cela est facilité dans un langage de programmation fonctionnel.

Les langages de programmation fonctionnels

Dans un sens restreint, un langage de programmation fonctionnel (LPF) est un langage de programmation sans variables modifiables, affectations ou structures de contrôle impératives.

Dans un sens plus général, un langage de programmation fonctionnel est un langage qui facilite l'écriture de programmes élégants avec les fonctions.

En particulier, les fonctions dans un LPF sont *des individus de première classe*. Cela signifie que :

- Comme les autres données, elles peuvent être définies à l'intérieur d'autres fonctions.
- Comme les autres valeurs, elles peuvent être passées en paramètres et retournées comme résultat.
- Comme pour d'autres valeurs, il existe un ensemble d'opérations de base qui permettent de composer les fonctions.

Quelques Langages de Programmation Fonctionnels

Dans son sens restreint :

- Pure Lisp, XSLT, XPath, XQuery
- Haskell (sans la monade I/O)

Dans son sens plus général :

- Lisp, Scheme,
- ML, Haskell,
- Pizza, Scala,
- Smalltalk(!),
- Java(?)

Histoire des LPF

1960	Lisp
1975–77	ML, FP, Scheme,
1978	Smalltalk
1986	Standard ML
1990	Haskell
1990	Erlang
1996	Caml
1996	Pizza
1997	Java 1.1
1999	XSLT, XPath
2000	OCaml
2003	XQuery

Livres recommandés

- Structure and Interpretation of Computer Programs. Harold Abelson and Gerald J. Sussman. 2nd edition. MIT Press 1996.
- Approche Fonctionnelle de la Programmation. Guy Cousineau et Michel Mauny. Ediscience International. 1998.
- Introduction to Functional Programming using Haskell. Richard Bird. Prentice Hall 1998.
- Foundations of Logic Programming. J.W. Lloyd. Springer Verlag. 1984.
- Effective Java. Joshua Bloch. Addison Wesley, 2001.
- Scala By Example. Martin Odersky. Disponible à *http://scala.epfl.ch/docu*.

De nombreux exemples et exercices du cours sont tirés du livre de Abelson et Sussman.

Mais plutôt que Scheme on utilise...

Scala

Dans ce cours on enseigne la PF avec un nouveau langage: Scala.

Scala possède toutes les constructions d'un LPF classique, comme :

- Les fonctions de première classe,
- le filtrage de motifs (pattern matching),
- un typage sûr, statique et flexible.

Scala est aussi un langage orienté-objet (pur).

Scala interopère simplement avec Java.

Il est particulièrement adapté pour le calcul symbolique, comme par exemple le traitement de documents XML.

Exemple : requêtes XPath

- Voici un petit exemple démontrant l'intérêt de la programmation fonctionnelle pour XML.
- Cet exemple utilise plusieurs caractéristiques de Scala qui seront expliquées une par une en détail dans les prochains cours.
- Un document XML est conceptuellement un arbre.
- Chaque noeud de l'arbre contient une étiquette (une chaîne de caractères), et zéro ou plusieurs sous-arbres.
- On peut représenter de tels arbres par une classe comme :

```
case class Tree(label: String, children: Tree*);
```

Considérons un petit document XML (en fait HTML).

```
<html>
  <head>
    <title>
      XML Programming in Scala
    </title>
  </head>
  <body>
    <h1>
      XML Programming in Scala
    </h1>
    <b>
      visit
      <a> scala.epfl.ch </a>,
      today!
    </b>
  </body>
</html>
```

En utilisant la classe *Tree*, on peut représenter ce document par :


```
val doc =
  Tree("html",
    Tree("head",
      Tree("title",
        Tree("XML Programming in Scala")
      )
    ),
    Tree("body",
      Tree("h1",
        Tree("XML Programming in Scala")
      ),
      Tree("b",
        Tree("visit"),
        Tree("a", Tree("scala.epfl.ch")),
        Tree("today!")
      )
    )
  );
```

(La nouvelle version de Scala permet aussi la syntaxe XML standard).

Expressions de chemin

On peut extraire des noeuds d'un document XML avec des expressions de chemin. Elles sont définies (inductivement) comme suit :

- *<string>* est une expression de chemin.
- Si *p* est une expression de chemin, alors
  <string> / *p* et
  <string> // *p* sont aussi des expressions de chemin.

Une expression de chemin peut filtrer un arbre, donnant comme résultat une liste de noeuds.

- *<string>* filtre un noeud étiqueté par *<string>*, et renvoie ce noeud.
  <string> / *p* filtre n'importe quel fils d'un noeud étiqueté par *<string>* qui est filtré par *p*.
  <string> // *p* filtre n'importe quel descendant d'un noeud étiqueté par *<string>* qui est filtré par *p*.

Remarque : les descendants sont les fils, petit-fils, petit-petit-fils, etc.

Example: *"html//a"* extrait tous les éléments étiquetés par "a" dans un document html, où qu'ils puissent apparaître.

Requêtes avec des expressions de chemin

Nous aimerions faire des recherches dans un document XML en utilisant des expressions XPath. C'est ce que fait la fonction *search*, qui prend en entrées un document XML et une expression de chemin.

La fonction doit retourner tous les sous-arbres du document XML qui sont filtrés par l'expression de chemin.

Example:

- *search(doc, "html//body//a")* doit retourner la liste constituée uniquement du sous-arbre :

  Tree(a,Tree(scala.epfl.ch))
- *search(doc, "html/body/a")* doit retourner la liste vide, pour signaler l'absence de correspondance.
- *search(doc, "html//XML Programming in Scala")* doit retourner la liste constituée de deux fois *Tree(XML Programming in Scala)*.

Implantation de XPath

Les requêtes XPath sont implantées par deux fonctions mutuellement récursives, *search* et *searchList*. Elles sont définies comme suit :

```
def search(tree: Tree, path: String): List[Tree] = split(path) match {
  case Pair(tree.label, rest) ⇒
    if (rest startsWith "//")
      searchList(descendants(tree), rest substring 2)
    else if (rest startsWith "/")
      searchList(children(tree), rest substring 1)
    else
      List(tree)
  case _ ⇒
    List()
}
def searchList(trees: List[Tree], path: String): List[Tree] =
  trees.flatMap(tree ⇒ search(tree, path));
```


Remarques :

1. La fonction *split* coupe une chaîne de caractères à la première occurrence du caractère /. En utilisant des méthodes de *java.lang.String*, elle peut être définie ainsi :

```
def split(s: String): Pair[String, String] = {
  val pos = s indexOf '/';
  if (pos == −1) Pair(s, "")
  else Pair(s.substring(0, pos), s.substring(pos))
}
```


2. On décompose le résultat de *split* par *filtrage de motif* (pattern matching), une généralisation de l'instruction *switch* en Java.

- Si le premier élément du chemin correspond à l'étiquette de l'arbre, alors
  - Si le chemin continue par //, faire une recherche sur tous les descendants de l'arbre avec le chemin restant après //,
  - sinon, si le chemin continue par /, faire une recherche sur tous les fils de l'arbre avec le chemin restant après /,
  - sinon, si le chemin s'arrête avec l'étiquette, retourner l'arbre lui-même.

  Si le premier élément du chemin ne correspond pas à l'étiquette de l'arbre, retourner la liste vide.

3. La fonction *search* utilise la méthode Java *substring*, ainsi que les fonctions *children* et *descendants* sur les arbres. Celles-ci sont définies ainsi :

$\textbf{\textit{def}}\ children\,(tree\colon\ Tree)\colon\ List\,[Tree] =$
$\quad tree.children.asInstanceOf[List\,[Tree]];$

$\textbf{\textit{def}}\ descendants\,(tree\colon\ Tree)\colon\ List\,[Tree] =$
$\quad children\,(tree)\ :::\ children\,(tree).flatMap\,(descendants);$

C'est tout le code dont on a besoin.

Les éléments qui ont été utilisés sont :

- Des définitions de valeurs non modifiables
- Des définitions de fonctions (récursives),
- Des fonctions comme arguments d'autres fonctions
- Le filtrage de motif

Les éléments dont on a pu se passer sont :

- Les variables modifiables
- Les affectations
- Les boucles

C'est ce qui caractérise la programmation fonctionnelle.

Eléments de Programmation

N'importe quel langage de programmation non-trivial possède :

- Des expressions primitives, qui représentent les entitées les plus simples manipulées par le langage.
- Un moyen de combiner, par lequel les éléments sont construits à partir d'éléments plus simples.
- Un moyen d'abstraire, par lequel les éléments sont nommés et manipulés en tant qu'unité.

Les expressions

La programmation fonctionnelle c'est un peu comme utiliser une calculatrice.

L'utilisateur tape des expressions et le système répond en calculant leur valeur.

Exemple:

$?\ 87 + 145$
232

$?\ 1000 - 333$
667

$?\ 5 + 2 * 3$
11

L'environnement

Les langages de programmation fonctionnels sont plus riches que les calculatrices, car ils permettent aussi de définir de nouvelles quantités et de nouveaux combinateurs.

Exemple:

$?\ \textbf{\textit{def}}\ size = 2$
$\textbf{\textit{def}}\ size\colon Int$

$?\ 5 * size$
10

$?\ \textbf{\textit{def}}\ pi = 3.14159$
$\textbf{\textit{def}}\ pi\colon Double$

$?\ \textbf{\textit{def}}\ radius = 10$
$\textbf{\textit{def}}\ radius\colon Int$

$?\ 2 * pi * radius$
62.8318

? ***def*** *circumference* $= 2 * pi * radius$
def *circumference*: *Double*

? *circumference*
62.8318

Le système de PF garde en mémoire les définitions dans un environnement, qui est une table de paires nom/valeur.

Evaluation

Une expression composée est évaluée de façon répétée comme suit :

- prendre l'opération la plus à gauche
- évaluer ses opérandes
- appliquer l'opérateur aux valeurs des opérandes.

Un nom défini est évalué en remplaçant le nom par la partie droite de sa définition.

Le processus d'évaluation s'arrête une fois qu'on a atteint une valeur.

Une valeur est un nombre (pour l'instant).

Plus tard nous serons amené à considérer des types de valeurs plus compliqués.

Exemple

Voici l'évaluation d'une expression arithmétique.

$$\begin{array}{ll} & (2 * pi) * radius \\ \rightarrow & (2 * 3.14159) * radius \\ \rightarrow & 6.28318 * radius \\ \rightarrow & 6.28318 * 10 \\ \rightarrow & 62.8318 \end{array}$$

Le processus de simplification pas-à-pas des expressions en valeurs est appelé réduction.

Paramètres

Les définitions peuvent avoir des paramètres. Exemples :

? ***def*** *square* (x: *Double*) $= x * x$
def *square* (x: *Double*): *Double*

? *square* (2)
4.0

? *square* (5 + 4)
81.0

? *square* (*square* (4))
256.0

? ***def*** *sumOfSquares* (x: *Double*, y: *Double*) = *square* (x) + *square* (y)
def *sumOfSquares* (x: *Double*, y: *Double*): *Double*

Evaluation des fonctions

Les applications de fonctions avec paramètres sont évaluées de la même manière que les opérations :

- Evaluer tous les arguments de la fonction (de gauche à droite).
- Remplacer l'application de fonction par la partie droite de la fonction, et dans le même temps
- Remplacer tous les paramètres formels de la fonction par les arguments effectifs.

Exemple:

```
    sumOfSquares (3, 2+2)
→
    sumOfSquares (3, 4)
→
    square (3) + square (4)
→
    3 * 3 + square (4)
→
    9 + square (4)
→
    9 + 4 * 4
→
    9 + 16
→
    25
```

On appelle cela la modèle de calcul par substitution.

Appel-par-Valeur et Appel-par-Nom

L'interpréteur réduit les arguments des fonctions vers des valeurs avant de réécrire l'application de fonction.

On pourrait aussi, à la place, appliquer la fonction à des arguments non réduits.

Exemple:

```
    sumOfSquares (3, 2+2)
→
    square (3) + square (2+2)
→
    3 * 3 + square (2+2)
→
    9 + square (2+2)
→
    9 + (2+2) * (2+2)
→
    9 + 4 * (2+2)
→
    9 + 4 * 4
→
    9 + 16
→
    25
```

Le deuxième ordre d'évaluation est appelé appel-par-nom, alors que le premier est appelé appel-par-valeur.

Pour les expressions qui utilisent seulement des fonctions pures et qui peuvent par conséquent être réduites par le modèle de substitution, les deux schémas retournent les mêmes valeurs finales.

L'appel-par-valeur a l'avantage d'éviter l'évaluation répétée des arguments.

L'appel-par-nom a l'avantage d'éviter l'évaluation des arguments quand le paramètre n'est pas utilisé du tout par la fonction.

L'évaluation en appel-par-nom termine toujours en une valeur si l'appel-par-valeur termine en une valeur.

L'inverse n'est pas vrai !

Si on considère :

```
? def loop: Int = loop
def loop: Int
```

```
? def first(x: Int, y: Int) = x
def first(x: Int, y: Int): Int
```

Alors avec

l'appel-par-nom	*l'appel-par-valeur*
first(1, loop)	first(1, loop)
→ 1	→ first(1, loop)
	→ first(1, loop)
	→ ...

Scala utilise l'appel-par-valeur sauf quand le paramètre est précédé de ***def***.

Exemple:

```
? def constOne(x: Int, def y: Int) = 1
constOne(x: Int, def y: Int): Int
```

```
? constOne(1, loop)
1
```

```
? constOne(loop, 2)                    (boucle infini)
^C
```


Expressions Conditionnelles et Prédicats

Scala offre une expression ***if–else*** pour exprimer un choix entre deux alternatives.

Sa syntaxe est comme celle du ***if–else*** de Java, mais son utilisation est comme celle de l'expression conditionnelle ... ? ... : ... de Java.

Exemple:

```
? def abs(x: Double) = if (x ≥ 0) x else −x
abs(x: Double): Double
```

$x \geq y$ est un prédicat de type *Boolean*.

Les expressions booléennes sont composées des constantes *true* et *false*, des opérateurs de comparaison, de la négation booléenne *!* et des opérateurs booléens && et ||.

Exercice: Définir && et || en termes de ***if***.

Exemple: Racines carrées par la méthode de Newton

Travail: Ecrire une fonction

```
def sqrt (x: Double): Double = ... // calcule la racine carrée de 'x'.
```

La manière la plus classique de calculer les racines carrées est par la méthode de Newton consistant en approximations successives :

- Commencer avec une estimation initiale y (disons: $y = 1$).
- Améliorer l'estimation en prenant la moyenne de y et x/y.

Exemple: *sqrt (2)*

Estimation	*Quotient*	*Moyenne*
1	2/1 = 2	1.5
1.5	2/1.5 = 1.3333	1.4167
1.4167	2/1.4167 = 1.4118	1.4142
1.4142	...	...

Implantation en Scala

1. Définir une fonction qui itère d'une estimation vers un résultat :

```
def sqrtIter (guess: Double, x: Double): Double =
  if (isGoodEnough (guess, x)) guess
  else sqrtIter (improve (guess, x), x)
```

Remarquons que *sqrtIter* est récursive.

Les fonctions récursives doivent être pourvues d'un type de retour explicite en Scala.

Pour les fonctions non récursives le type de retour est optionnel: s'il est manquant, il est calculé à partir de la partie droite de la fonction.

2. Définir une fonction pour *améliorer* l'estimation et un test de terminaison *isGoodEnough*.


```
def improve (guess: Double, x: Double) =
  (guess + x / guess) / 2

def isGoodEnough (guess: Double, x: Double) =
  abs (square (guess) − x) < 0.001
```

3. Définir la fonction *sqrt*.

```
def sqrt (x: Double) = sqrtIter (1.0, x)
```

Exercice: Le test *isGoodEnough* n'est pas très précis pour les petits nombres et peut ne pas terminer pour les très grands (pourquoi ?).

Concevoir une version différente de *isGoodEnough* qui n'a pas ces problèmes.

Exercice: Donner la trace de l'exécution de l'expression *sqrt (4)*.

Fonctions imbriquées

La programmation fonctionnelle encourage la construction de nombreuses petites fonctions d'aide.

Les noms des fonctions telles que *sqrtIter*, *improve* et *isGodEnough* n'ont d'intérêt que pour l'implantation de *sqrt*.

Normalement on ne voudrait pas que les utilisateurs de *sqrt* accèdent directement à ces fonctions.

On peut atteindre ce but (et éviter la pollution de l'espace de noms) en incluant les fonctions d'aide à l'intérieur même de la fonction appelante :

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double, x: Double): Double =
    if (isGoodEnough(guess, x)) guess
    else sqrtIter(improve(guess, x), x);

  def improve(guess: Double, x: Double) =
    (guess + x / guess) / 2;

  def isGoodEnough(guess: Double, x: Double) =
    abs(square(guess) - x) < 0.001;

  sqrtIter(1.0, x)
}
```

Remarques:

- { ... } délimite un bloc.
- La fin du bloc est une expression qui définit sa valeur.
- Cette expression peut être précédée par des définitions auxiliaires.
- Les définitions à l'intérieur d'un bloc ne sont visibles que dans le bloc.
- Les définitions à l'intérieur d'un bloc masquent les définitions des blocs englobants.

Enoncés et blocs

- Les définitions et les expressions sont deux sortes d'*énoncés*.
- Un *bloc* est composé d'une ou plusieurs définitions, suivies par une expression.
- L'expression finale définit le résultat du bloc.
- Les blocs sont eux-mêmes des expressions, ils peuvent apparaître partout où une expression le pourrait.
- Chaque définition dans un bloc doit être suivie d'un point virgule, *sauf* si la définition se termine par une accolade } et est suivie d'une nouvelle ligne.
- **Exemple:**

```
def f = (...)                   def f = { ... }
def g = ...                     def g = ...
^ error: ';' expected.          // OK.
```


Portée lexicale

Les noms définis dans les blocs englobants sont aussi visibles dans les blocs internes, pourvu qu'ils n'y soient pas masqués.

Par conséquent, on peut simplifier notre exemple *sqrt* en omettant de passer *x* aux fonctions internes.

```
def sqrt(x: Double) = {
  def sqrtIter(guess: Double): Double =
    if (isGoodEnough(guess)) guess
    else sqrtIter(improve(guess));

  def improve(guess: Double) =
    (guess + x / guess) / 2;

  def isGoodEnough(guess: Double) =
    abs(square(guess) - x) < 0.001;

  sqrtIter(1.0)
}
```