Résumé de Scala

Semaine

Page

- 1 -

Appel-par-valeur 39

Utilisé par défaut. Evalue tout de suite les paramètres.

Avantage : Evite l'évaluation répétée des paramètres au fil de l'exécution de la fonction

Appel-par-nom

La fonction doit être définie de la façon suivante : myfunc(x : => int)

Avantage : Evite d'évaluer le paramètre s'il n'est jamais utilisé

Fonction récursive et type de retour 46

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

- 2 -

Récursivité terminale 4

Fonction dont la valeur retournée est directement la valeur obtenue par un appel récursif, sans qu'il n'y ait aucune opération sur cette valeur

non-terminale :
def factorial(n : Int): Int = if (n == 0) 1 else n * factorial(n – 1)
terminale : (grâce à l'accu)
def factorial (n : Int, accu : Int): Int = if (n < 1) accu else factorial(accu * n, n – 1)

‘def’ 6

Syntaxe : def f = expr
Particularité : expr évalué à chaque fois que f sera utilisée

‘val’

Syntaxe : val x = expr
Particularité : expr évalué lors de la définition uniquement

Fonction d'ordre supérieur 7

Fonction qui prend d’autres fonctions en paramètres ou qui retourne une fonction en résultat

Exemple : def sum(f : Int ⇒ Double, a : Int, b : Int): Double = if (a > b) 0 else f(a) + sum(f, a + 1, b)

Fonction anonyme 10

Syntaxe : (x1 : T1 , ..., xn : Tn ⇒ E)
Avantage : notation plus courte, idéal pour les fonctions à usage unique
Equivalent à : { def f (x1 : T1 , ..., xn : Tn) = E ; f }

Fonction currifiée 13

La fonction currifiée d'une fonction f est une fonction qui prend les premiers paramètres de f en argument et qui retourne une fonction qui traite les arguments de f non encore traités.

Syntaxe :

Définition : def f (args1) ... (argsn−1) (argsn) = (args1 ⇒ (args2 ⇒ ... (argsn ⇒ E) ...))
Création d'une fonction intermédiaire : def g (args2)...(argsn) = def f (args1) ... (argsn)
Récupération d'une valeur : f ("hello") (1, 10) (...)

- 3 -

Classe 1

Définition : class Rational(x : int, y : int) { def n = x; def d = y}
Définit à la fois un nouveau type et un constructeur
Création : new Rational (1, 2)

Accès aux éléments : x.n
Méthode : def add(r : Rational) = new Rational(n * r.d + r.n * d, d * r.d)
Attribut des 'def' : private, protected, override (redéfinit une méthode héritée)
Attribut des 'class' : abstract (classe non instanciable), extends (par défaut extend scala.ScalaObject)
Auto-référence : this représente l'objet dont on exécute la méthode
Constructeur primaire : constructeur définit lors de la définition de la classe
Constructeur auxiliaire : def this(x : int) = this(x, 1) (constructeur à un argument)
Classes standards : les types de bases sont en fait des classes (par alias : type int = scala.Int)

Abstraction de données

Possibilité de choisir plusieurs implémentations des données sans affecter les clients

Méthode et Opérateur infixe 16

Toute méthode à un paramètre peut être utilisée comme un opérateur infixe

Exemple : r.add(x) => r add x ou encore r.+(x) => r + x

L'opérateur est soit une lettre suivi de lettres ou de chiffres, soit un ensemble de symboles d'opérateurs

Evaluation et Priorité 17

(toutes les lettres) | ^ & < > = ! : + - */% (tous les autres caractères spéciaux)

Associatif à gauche hormis les opérateurs terminant par ':'

Classe cas 7

Définition de classe normale, définit implicitement des fonctions de construction

Exemple :

```
abstract class Expr // définit une classe abstraite
    case class Number(n : int) extends Expr // et deux sous-classes (et leur constructeur)
    case class Sum(e1 : Expr, e2 : Expr) extends Expr
```

Avantage : Permet (facilement ?) le filtrage de motif sur le type de la classe

Filtrage de motif 8

Syntaxe : e match { case p_1 ⇒ e_1 ... case p_n ⇒ e_n }

p soit constructeur ($C(p_1,...,p_n)$), soit constante c,
soit variable x (possibilité de spécifier le le type : 'x: String')
retourne MatchError si aucun 'case' ne convient
possibilité de mettre des jokers (wildcard) _

Fonctions polymorphes / Classes paramétriques 21

Fonctions / Classes dont le type des arguments est passé comme paramètres de type

Syntaxe : def length[a](xs : List[a]): int = ... / abstract class List[a] {...}
Appel : length[Int](xs) (on peut les omettre quand inférés par les paramètres)

Erreur 25

Syntaxe : error("my msg"), fonction retournant 'Nothing', c-à-d ne retournant rien

Tuple 36

Tuple et motif : case Tuple2(n, d) ou plus simplement case (n, d)
Alias : (x1 , ..., xn) alias de Tuple*n*(x1 , ..., xn), (T1 , ..., Tn) alias de Tuple*n*[T1 , ..., Tn]

Liste 27

Les éléments d'une liste sont homogènes

Type : List[T]
Liste et motif : '::' et Nil sont des classes cas, possibilité d'utiliser des motifs de filtrage
Cas fréquents : x :: Nil (équivalent à List(x)), x :: xs, Nil (équivalent à List())
Opération :
'::' concatène l'élément de gauche avec la liste de droite, retourne la liste obtenue
associatif à droite : 1 :: 2 :: 3 :: Nil équivaut à 1 :: (2 :: (3 :: Nil))
':::' concatène deux listes
head : retourne le 1er élément de la liste, retourne la liste obtenue
tail : retourne la liste amputée de son premier élément
isEmpty : retourne true si la liste est vide
length : retourne la longueur de la liste
last : retourne le dernier élément de la liste
init : retourne la liste amputée de son dernier élément
reverse : renverse les éléments d'une liste
take(n) : retourne les n premiers éléments de la liste
drop(n) : retourne les éléments de la liste amputée de ses n premiers éléments
apply(n) : retourne le n-ième élément de la liste
map(f : a ⇒ b): List[b] : applique la fonction f à tous les éléments de la liste
filter(p) : renvoie une liste dont les éléments respectent p

reduceLeft(op : (a, a) ⇒ a): a : insère un opérateur binaire entre deux éléments adjacents
foldLeft(z : a)(op : (b, a) ⇒ b): b : similaire à reduceLeft, il ajoute au début l'élément z
reverse avec foldLeft :
def reverse[a](xs : List[a]): List[a] = (xs foldLeft List[a]()){(xs, x) ⇒ x :: xs}
reduceRight et foldRight : similaire à leur équivalent left, hormis sens des opérations inversé
range(d, f) : crée une liste d :: d + 1 :: d + 2 :: .... :: f – 1 :: Nil
flatMap(f : a ⇒ List[b]): List[b] : = map hormis le prototype de la fonction en paramètre
forall(p) : teste si tous les éléments de la liste respectent p
find(p) : retourne le premier élément satisfaisant p
count(p) : compte le nombre d'éléments de la liste qui satisfait le prédicat
exists(p) : teste si au moins un élément dans la liste satisfait p
zip(L) : combine deux listes en une liste de couple
zipAll(L, a, b) : comme zip mais complète avec a (si this.length > L.length) ou b

Induction structurelle 20

Cas de base : P(List()) vrai ?
Etape d'induction : si P(xs) vrai, alors P(x :: xs) vrai ?
Exemple : xs.reverse.reverse = xs
Cas de base : List().reverse.reverse = List().reverse = List()
Etape d'induction : (la plupart des parenthèses ne sont la que pour améliorer la lisibilité

(x :: xs).reverse.reverse = (xs.reverse ::: List(x)).reverse	x :: xs = x :: (xs.reverse.reverse)

Il reste à prouver (xs.reverse ::: List(x)).reverse = x :: (xs.reverse.reverse)
Faisons la substitution suivante : xs.reverse => ys
(ys ::: List(x)).reverse = x :: (ys.reverse)
De nouveau, induction :
Cas de base :

(List() ::: List(x)).reverse = List(x).reverse = List(x)	x :: (List().reverse) = x :: List() = List(x)

Etape d'induction :

((y :: ys) ::: List(x)).reverse = (ys ::: List(x)).reverse ::: List(y)	x :: ((y :: ys).reverse) = x :: (ys.reverse) ::: List(y)

Sur les arbres :
Cas de base : P(l) vrai pour toutes les feuilles l ?
Etape d'induction : pour chaque noeud interne t avec sous-arbre s1,...,sn,
P(s1) ∧ ... ∧ P(sn) ⇒ P(t) vrai ?

Une moyen de prouver une implantation consiste à prouver des lois qu'elle respecte.

Notation For 1

Syntaxe : for (s) yield e ou **s** est une séquence de **générateurs** et de **filtres**,
et **e** l'élément de la liste de retour
Filtre : expression de type booléan, écarte les liaisons pour lesquelles f est faux
Générateur : val p ← e', lie les variables dans le motif p aux valeurs successives de la liste
Doit commencer par un générateur. Si plusieurs, les derniers varient plus rapidement
Exemple 1 : Soit une liste de persons (objet (~structure de donnée) avec champ name et age)
For : for (val p ← persons; p.age > 20) yield p.name
Fonctions : persons filter (p ⇒ p.age > 20) map (p ⇒ p.name) //renvoie une List[String]
Exemple 2 : for (val i ← List.range(1, n); val j ← List.range(1, i); isPrime(i+j)) yield (i, j)
Généralisation : Il suffit que la structure de donnée implémente map, flatMap et filter car

for (val x ← e) yield e'	e.map(x ⇒ e')
for (val x ← e; f; s) yield e'	for (val x ← e.filter(x ⇒ f); s) yield e'
for (val x ← e; val y ← e'; s) yield e"	e.flatMap(x ⇒ for (val y ← e'; s) yield e")

Classe Anonyme 10

Agit comme une définition d'une classe locale et une valeur de cette classe

Exemple :

```
class Book { val title : String ; val authors : List[String] }
val books : List[Book] = List( new Book {
                                val title = "Structure and Interpretation of Computer Programs"
                                val authors = List("Abelson, Harald", "Sussman, Gerald J.") })
```


Fonction et objet 1

Concept :
Langage fonctionnel : fonction = valeur, langage orienté objet : valeur = objet ⇒ fonction = objet
Fonction instance du trait Function*n*[a1 , ..., an , b] { def apply(x1 : a1 , ..., xn : an): b }
Conséquence : x(y1 , ..., yn) est un raccourci pour x.apply(y1 , ..., yn)

Alias : (T1 , ..., Tn) ⇒ U alias de Function*n*[T1 , ..., Tn , U]

Classe abstraite

Contient des méthodes sans implémentation, ne peut pas être instancié

Trait : Similaire aux interfaces java, peut par contre être partiellement implémenté, aucun paramètres constructeur

Fonction de filtrage anonyme 5

Exemple : Les deux sont équivalents

```
xss flatMap { case x :: xs ⇒ List(x) ; case List() ⇒ List() } // version anonyme
xss.flatMap { y ⇒ y match { case x :: xs ⇒ List(x) ; case List() ⇒ List() }}
```

Décomposition orientée objet 20

Avantage : ajout de nouvelles classes faciles

Désavantage : ajout de méthode pénible, implique l'ajout d'une nouvelle méthode dans chaque sous-classe existante

Exemple :

```
abstract class Expr {
        def eval: Int
        }
class Number(n: Int) extends Expr {
        def eval: Int = n
        }
class Sum(e1: Expr, e2: Expr) extends Expr {
        def eval: Int = e1.eval + e2.eval
        }
```

Décomposition utilisant le filtrage de motifs 21

Avantage : utilisation des classes cas et du filtrage de motifs, ajout de nouvelles méthodes faciles, processus de construction renversé par les fonctions de tests et d'accès

Désavantage : l'ajout d'un nouveau type d'expression implique de retrouver toutes les expressions utilisant le filtrage de motifs pour ajouter le nouveau cas

Exemple :

```
abstract class Expr // Utilisation de classes cas
        case class Number(n : int) extends Expr
        case class Sum(e1 : Expr, e2 : Expr) extends Expr

def eval(e : Expr): int = e match { // avec filtrage de motifs
        case Number(n) ⇒ n
        case Sum(e1, e2) ⇒ eval(e1) + eval(e2)
        }
```

- 8 -

Objets à états 2

Uniquement si l'objet change avec le temps

Avantage : utilisation possible des effets de bord (via notamment par le passage par référence des objets)

'var' 3

Syntaxe : var r = expr

Particularité : peut être modifié contrairement à val et def

Equivalence opérationnelle 8

- Exécuter les définitions de x et y
- Exécuter une séquence arbitraire d'opérations qui impliquent x et y
- Réexécuter cette séquence en intervertissant x et y
- Si les résultats obtenus dans les deux précédentes exécutions ne sont pas égaux, x n'est pas équivalent à y. Par contre, si pour toutes les séquences, les résultats sont les mêmes, x est équivalent à y.

Boucle 12

While : est défini comme suit

def while(condition : ⇒ boolean)(command : ⇒ unit): unit =

if (condition) { command; while(condition)(command) }

Type 22

Il est possible de définir des types

Exemple : type Action = () ⇒ unit

- 10 -

Calcul avec flots 1

On construit une liste qui contiendra les valeurs successives

Stream 5

Pas de filtrage de motifs, même interface que les listes hormis empty remplace Nil, cons remplace ::, append remplace :::

Evaluation retardée : Stream calcule au fur et à mesure ses éléments lors de la demande

'lazy'

Syntaxe : lazy val tail

Avantage : Evalué seulement lors du 1er appel

Itérateur 28

représente une suite d'éléments qui sont évalués à la demande

next() : renvoie l'élément actuel, et avance d'un pas

hasNext() : renvoie vrai s'il y a encore un élément

Avantage : plus rapide que les flots, implanté aussi dans les langages primaires généralement

Désavantage : dépendance d'ordre : les éléments arrivent dans un ordre donné, pas de réutilisabilité : un élément est perdu après être appelé

-9-

Contrainte

permet de résoudre des équations mathématiques notamment
Composé de contraintes et de quantités 7

- **Quantity** : class représentant les entités mathématiques (dans notre cas) 8
  - getValue : retourne la valeur actuelle de la quantité
  - setValue : donne la valeur
  - forgetValue : supprime la valeur
  - connect : déclare que la quantité participe à une contrainte
- **Constraint** : classe représentant les relations entre les entités mathématiques 12
  - newValue : appelé lorsque une quantitée connectée reçoit une nouvelle valeur
  - dropValue : appelé lorsque une quantité connectée perd sa valeur
- Si réveillé par newValue, essaiera de calculer les quantités auxquelles elle est connectée
- **Constante** : contrainte qu'on ne peut redéfinir ou "oublier" 16
- **Sonde** : contrainte qui affiche les changements de la quantité attachée 21

Option 9

Défini ainsi :

```
trait Option[+A]
case class Some[+A](value : A) extends Option[A]
case object None extends Option[Nothing] // nothing : sous-type de n'importe quel autre type
```

Covariance 10

Option[+A] signifie que Option est un constructeur de type co-variant :
si T est un sous-type de S (noté T <: S), alors Option[T] est un sous-type de Option[S]

Amélioration 22

```
val C, F = new Quantity # ===, + et * méthodes de la classe quantity
C * c(9) === (F + c(-32)) * c(5) # === construit une contrainte, c fonction retournant une quantité
```

-11-

Résumé Lisp

Expression 4

Une expression Lisp composée est une séquence de sous-expressions entre parenthèses
La première sous-expression d'une expression composée dénote un opérateur
Les autres expressions dénotent les opérandes
Type des données: nombres (entier ou flottant), chaînes, symboles (chaînes sans apostrophes) et listes

Formes spéciales 5

define name expr : définition d'alias
lambda (params) expr : la fonction anonyme qui prend params et renvoie expr
if cond expr1 expr2 : similaire au if traditionnel
quote (list) ou '(list) : empêche l'évaluation de la liste

Liste 9

Exemple : (1.0 "hello" (1 2 3))
nil : liste vide (~ Nil)
cons x y : colle la tête x à la queue y (~ ::)
null? x : retourne vrai si la liste x est vide (~ isEmpty)
car x : retourne la tête de la liste (~ head)
cdr x : retourne la queue de la liste (~ tail)

-13-

Résumé Prolog

Clauses 5

Faits : Etablit que Xs concaténé à Ys donne Zs

Syntaxe : ma_procedure (Xs, Ys, Zs).

Règle : Etablit que Xs concaténé à Ys donne Zs pour autant que Us concaténé à Vs donne Ws

Syntaxe : ma_procedure (Xs, Ys, Zs) :- ma_procedure (Us, Vs, Ws).

Comportement

append([], Ys, Ys). 4

append([X |Xs], Ys, [X |Zs]) :– append(Xs, Ys, Zs).

append(X, [2, 3], [1, 2, 3]) renvoie X = [1] 6

same(X, X). 11

sibling(X, Y) :– child(X, Z), child(Y, Z), not(same(X, Y)). //définition d'une nouvelle règle

Requète 8

Recherche dans la base de donnée les entrées satisfaisant le prédicat

Syntaxe : predicat(X, Y)?

more : demande de solutions supplémentaires pour la requète précédente 9

Opérateurs

Concaténation de liste : [X | Xs]

Inversion : not 11

exemple : male(X) :– not(female(X)).

Règles récursives 12

Exemple : définir que X est un ancètre de Y

parent(X, Y) :– child(Y, X).

ancestor(X, Y) :– parent(X, Y).

ancestor(X, Y) :– parent(X, Z), ancestor(Z, Y).

Sucres syntaxiques pour les listes 15

[]	= nil
[S |T]	= cons(S, T)
[S]	= cons(S, nil)
[T1 , ..., Tn]	= cons(T1 , ... cons(Tn , nil) ...)