

Partie IVa : Arbres de syntaxe abstraite en Scala

Martin Odersky, LAMP/DI

1

Arbres de syntaxe abstraite en Scala

- Scala permet une représentation directe des arbres de syntaxe abstraite grâce aux classes cas.
- Exemple:

```
package calc with {
  abstract class Tree;
  abstract case class NumLit(value: Int) extends Tree;
  abstract case class Operation(
    op: Int, left: Tree, right: Tree) extends Tree;
}
```
- Cela combine les avantages de la décomposition orientée-objet et des visiteurs.
 - Comme avec la décomposition OO, il est possible de déclarer des méthodes dans la classe de base qui sont ensuite redéfinies dans les sous-classes.
 - Comme avec les visiteurs, il est possible de reconnaître une sous-classe de façon très concise.

Martin Odersky, LAMP/DI

2

Construction d'arbres en Scala

- Les arbres sont, pour l'essentiel, construits de la même manière qu'en Java.
- Le mot-clé `new` est optionnel en Scala. Par ex.

```
new Operation(op, left, right)
ou
Operation(op, left, right)
```

- Voici un petit analyseur syntaxique pour des expressions :

Martin Odersky, LAMP/DI

3

Analyseur syntaxique d'expressions en Scala

```
package calc with {
import java.io;

class Parser(in: InputStream)
  extends Scanner(in) with {
  nextSym();

  def expression(): Tree = {
    var t = term();
    while (sym == PLUS ||
           sym == MINUS) {
      val startpos = pos;
      val operator = sym;
      nextSym();
      t = Operation (startpos,
                    operator, t, term());
    }
  }

  def term(): Tree = {
    var t = factor();
    while (sym == MUL || sym = DIV) {
      val startpos = pos;
      val operator = sym;
      nextSym();
      t = Operation (startpos,
                    operator, t, factor());
    }
  }
}

def factor(): Tree = {
  var t = null;
  if (sym == NUMBER) {
    t = NumLit (pos,
               Integer.parseInt(chars));
    nextSym();
  } else if (sym == LPAREN) {
    nextSym();
    t = expression();
    if (sym == RPAREN) nextSym();
    else error ("')' expected");
  } else {
    error("illegal start of " +
          "expression");
  }
}
```

Martin Odersky, LAMP/DI

4

Un visiteur d'arbres toString en Scala

- Les arbres peuvent être traversés grâce à la fonction match:

```
import calc;
def toString(tree: Tree): String = tree match {
  case NumLit(v) =>
    v.toString()
  case Operation(op, left, right) =>
    "(" + toString(left) + Scanner.representation(op)
      + toString(right) + ")"
}
```

Martin Odersky, LAMP/DI

5

Une réalisation de toString en utilisant la Décomposition OO

- Il est aussi possible d'écrire un processeur en utilisant la décomposition OO.
- Revoici toString, mais cette fois-ci sous la forme d'une méthode abstraite définie dans les différentes sous-classes.

```
package calc with {
  abstract class Tree with {
    abstract def toString(): String;
  }
  abstract case class NumLit(value: Int) extends Tree with {
    def toString() = String.valueOf(value)
  }
  abstract case class Operation(
    op: Int, left: Tree, right: Tree) extends Tree with {
    def toString() =
      "(" + toString(left) + Scanner.representation(op)
        + toString(right) + "«
  }
}
```

Martin Odersky, LAMP/DI

6

Un évaluateur d'expressions en Scala

```
import calc;

def eval(tree: Tree): Int = tree match {

  case NumLit(v: Value) =>
    v

  case Operation(op, left, right) =>
    val l = eval(left);
    val r = eval(right);
    op match {
      case Scanner.PLUS      => l + r
      case Scanner.MINUS    => l - r
      case Scanner.MUL      => l * r
      case Scanner.DIV      => l / r
    }
}
```

Martin Odersky, LAMP/DI

7

Classes cas et Extensibilité

- Les classes cas résolvent-elle le problème de l'extensibilité ?
- Presque, mais pas entièrement.
- Les classes cas laissent le choix entre la décomposition OO et les visiteurs.
- Un choix différent peut être fait pour chaque opération.
- Par exemple, on peut choisir la décomposition OO pour `toString` et les visiteurs pour `eval`.
- Mais, lorsqu'une dimension a été fixée, on doit s'y tenir.
- Par exemple, si l'on a choisi les visiteurs pour `eval` et que l'on ajoute ensuite de nouvelles sous-classes, on est obligé de modifier des expressions `match` existantes.
- Des solutions plus flexibles sont le sujet de recherches récentes.

Martin Odersky, LAMP/DI

8