

## Partie IV : Syntaxe abstraite

- Actions sémantiques
- Syntaxe abstraite
- Arbres de syntaxe abstraite
- Accéder aux arbres
- OO
- Décomposition orientée objet
- Visiteurs

Martin Odersky, LAMP/DI

1

## Actions sémantiques

- Un analyseur syntaxique fait généralement plus que simplement reconnaître une syntaxe.
- Il peut :
  - Évaluer du code (interpréteur)
  - Émettre du code (compilateur simple passe)
  - Construire une structure de données interne (compilateur multi-passes)
- De façon générale, un analyseur syntaxique réalise des *actions sémantiques*.
- Dans un analyseur à descente récursive, les actions sémantiques sont intégrées aux routines de reconnaissance.
- Dans un analyseur ascendant généré par une machine, elles sont ajoutées à la grammaire soumise au générateur d'analyseurs.

Martin Odersky, LAMP/DI

2

## Un analyseur / interpréteur

Le langage source : les expressions arithmétiques.

$$E = E "+" T \mid E "-" T \mid T$$
$$T = T "*" F \mid T "/" F \mid F$$
$$F = \text{number} \mid "(" E ")"$$

Le langage source dans une formulation LL(1) :

Martin Odersky, LAMP/DI

3

## Le programme du simple analyseur

```
package calc;
import java.io.*;
class Parser extends Scanner {
    public Parser(InputStream in) {
        super(in);
        nextToken();
    }
    public void expression() {
        term();
        while (token == PLUS ||
            token == MINUS) {
            nextToken();
            term();
        }
    }
    public void term() {
        factor ();
        while (token == MUL ||
            token == DIV){
            nextToken();
            factor();
        }
    }
    public void factor() {
        if (token == NUMBER) {
            nextToken();
        } else if (token == LPAREN) {
            nextToken();
            expression();
            if (token == RPAREN)
                nextToken();
            else error ("')' expected");
        } else
            error ("illegal start of " +
                "expression");
    }
}
```

Martin Odersky, LAMP/DI

4

## L'interpréteur

```
package calc;
import java.io.*;
class Parser extends Scanner {
    public Parser(InputStream in) {
        super(in);
        nextToken();
    }
    public int expression () {
        int v = term();
        while (token == PLUS ||
              token == MINUS) {
            int operator = sym;
            nextToken();
            int v2 = term();
            if (operator==PLUS) v = v + v2;
            else v = v - v2;
        }
        return v;
    }
    public int term() {
        int v = factor();
        while (token == MUL ||
              token == DIV) {
            int operator = sym;
            nextToken();
            int v2 = factor();
            if (operator==MUL) v = v * v2;
            else v = v / v2;
        }
        return v;
    }
    public int factor () {
        int v = 0;
        if (token == NUMBER) {
            v = Integer.parseInt(chars);
            nextToken();
        } else if (token == LPAREN) {
            nextToken();
            v = expression();
            if (token == RPAREN)
                nextToken();
            else error ("')' expected");
        } else
            error ("illegal start of " +
                  "expression");
        return v;
    }
}
```

Martin Odersky, LAMP/DI

5

## Arbres syntaxiques

- Dans un compilateur multi-passes, l'analyseur construit explicitement un arbre syntaxique.
- Toutes les phases suivantes du compilateur travaillent sur l'arbre de syntaxe abstraite, et non sur le programme source.
- Cet arbre peut être l'arbre de syntaxe concrète correspondant à la grammaire non contextuelle.
- Mais on utilise généralement une forme simplifiée.

Martin Odersky, LAMP/DI

6

## Syntaxe abstraite contre syntaxe concrète

Concernant l'arbre de syntaxe concrète, quelques simplifications sont possibles :

1. Pas besoin d'analyser un texte dans le langage abstrait, donc ( ) pas nécessaires.

`A * (B + C)` devient ...

2. Pas besoin de maintenir les symboles terminaux.

`If (x == 0) y = 1 else y = 2` devient ...

Martin Odersky, LAMP/DI

7

## Arbres de syntaxe abstraite

- Un arbre de syntaxe abstraite est un arbre avec un type de nœud pour chaque alternative dans la syntaxe abstraite.
- On représente un arbre en utilisant un ensemble de classes Java, une pour chaque alternative.
- Une super-classe abstraite commune : `Tree`.
- Chaque classe représente les sous-arbres comme variables d'instances.
- Chaque classe a un constructeur pour construire un nœud du type donné.

Martin Odersky, LAMP/DI

8

## Arbre de syntaxe abstraite (2)

- Exemple : Pour les expressions arithmétiques :

```
abstract class Tree {
    static class NumLit extends Tree {
        int value;
        ...
    }
    static class Operation extends Tree {
        int operator;
        Tree left, right;
        ...
    }
}
```

Martin Odersky, LAMP/DI

9

## Accéder aux arbres

- Les arbres de syntaxe abstraite constituent la structure de données d'entrée centrale des phases suivantes du compilateur.
- ⇒ Important de trouver une représentation qui puisse être utilisée de manière flexible.
- Comment les processeurs d'arbres accèdent-ils à cet arbre ?
- Une solution simple (et grossière); utiliser `instanceof` pour trouver le type du nœud syntaxique, puis le convertir pour accéder aux éléments de l'arbre, p.ex.

```
if (tree instanceof NumLit) {
    return ((NumLit)tree).value;
}
```

- Mais ceci n'est ni élégant ni efficace.
- Une meilleure solution : une décomposition orientée objet.
- Une solution encore meilleure : les *visiteurs*

Martin Odersky, LAMP/DI

10

## Exemple : Les expressions arithmétiques

- Nous allons présenter maintenant à la fois la décomposition orientée objet et l'accès par visiteur en utilisant l'exemple des expressions arithmétiques.
- Deux types de nœuds : `Operation`, `NumLit`.
- Deux types d'actions : `eval`, `toString`.
- Un exemple très simple.
- Typiquement les langages ont 20 (Misc), 40 (Java), ou plus types de nœuds.
- Un compilateur typique a 5 - 10 processeurs.
- Mais il n'y a qu'une différence d'échelle - la structure de base reste la même.

Martin Odersky, LAMP/DI

11

## Structure des classes

```
package calc;

public abstract class Tree {
    int pos; // position pour signaler
             // les erreurs, présente
             // dans tous les noeuds.

    /** une sous-classe pour
        représenter les nombres */
    static class NumLit extends Tree {
        int value;
        NumLit(int pos, int value) {
            this.pos = pos;
            this.value = value;
        }
    }

    /** une sous-classe pour
        représenter les opérations */
    static class Operation extends Tree {
        int operator;
        Tree left, right;
        Operation(int pos, int operator,
                  Tree left, Tree right)
        {
            this.pos = pos;
            this.operator = operator;
            this.left = left;
            this.right = right;
        }
    }
}
```

Martin Odersky, LAMP/DI

12

## Un analyseur qui construit un arbre

```
package calc;
import java.io.*;
import calc.Tree.*;

class Parser extends Scanner {
    public Parser(InputStream in) {
        super(in);
        nextToken();
    }

    public Tree expression() {
        Tree t = term();
        while (token == PLUS ||
              token == MINUS) {
            int startpos = pos;
            int operator = sym;
            nextToken();
            t = new Operation (startpos,
                              operator, t, term());
        }
        return t;
    }

    public Tree term() {
        Tree t = factor();
        while (token == MUL ||
              token == DIV) {
            int startpos = pos;
            int operator = sym;
            nextToken();
            t = new Operation (startpos,
                              operator, t, factor());
        }
        return t;
    }

    public Tree factor() {
        Tree t = null;
        if (token == NUMBER) {
            t = new NumLit (pos,
                           Integer.parseInt (chars));
            nextToken();
        } else if (token == LPAREN) {
            nextToken();
            t = expression();
            if (token == RPAREN)
                nextToken();
            else error ("')' expected");
        } else
            error ("illegal start of " +
                  "expression");
        return t;
    }
}
```

Martin Odersky, LAMP/DI

13

## Décomposition orientée objet

- Chaque processeur d'arbre  $P$  est représenté par une méthode dynamique  $P()$  dans chaque classe d'arbre.
- La méthode est abstraite dans la classe `Tree`, et implémentée dans chaque sous-classe.
- Pour traiter un sous-arbre, il suffit d'appeler sa méthode correspondant au processeur :  $t.P()$ .
- Dans notre exemple : définir les méthodes `eval` et `toString` dans les classes `NumLit` et `Operation`.
- Les méthodes `eval` et `toString` sont abstraites dans la classe `Tree`, donc elles peuvent être invoquées sur chaque arbre.
- Ce qu'elles font va dépendre du type concret de l'arbre.

Martin Odersky, LAMP/DI

14

## Décomposition OO pour les expressions

(on a omis les constructeurs)

```
package calc;

public abstract class Tree {
    int pos;

    public abstract String toString();
    public abstract int eval();

    static class NumLit extends Tree {
        int value;

        public String toString() {
            return String.valueOf(value);
        }

        public int eval() {
            return value;
        }
    }

    static class Operation extends Tree {
        int operator;
        Tree left, right;

        public String toString() {
            return "(" + left.toString() +
                Scanner.representation(
                    operator) +
                right.toString() + ")";
        }

        public int eval() {
            int l = left.eval();
            int r = right.eval();
            switch (operator) {
                case Scanner.PLUS:
                    return l + r;
                case Scanner.MINUS:
                    return l - r;
                case Scanner.MUL:
                    return l * r;
                case Scanner.DIV:
                    return l / r;
                default:
                    throw new InternalError();
            }
        }
    }
}
```

Martin Odersky, LAMP/DI

15

## Une classe « pilote »

```
package calc;

class Main {
    static public void main(String[] args) {
        System.out.print("> ");
        Tree t = new Parser(System.in).expression();
        if (t != null)
            System.out.println(t.toString() + " evaluates
                to " + t.eval());
    }
}
```

• Utilisation :

```
java calc.Main
> 2 * (3 + 4);
(2)*(3+4) evaluates to 14
```

Martin Odersky, LAMP/DI

16

## Extensibilité

- Avec un arbre de syntaxe abstraite, il peut y avoir extension dans deux dimensions.
  - Ajouter un nouveau type de nœud .
  - Ajouter un nouveau type de méthode de traitement.
- Quelle est la plus commune ?
- Quelle est la plus facile à réaliser ?
- Ajouter un nouveau type de nœud : ajouter une nouvelle sous-classe
- Ajouter une nouvelle méthode de traitement : ajouter des méthodes de traitement à chaque sous-classe.

Martin Odersky, LAMP/DI

17

## Visiteurs

- Le motif de conception (*design pattern*) "visiteur" permet une extension simple par de nouveaux traitements.
- Toutes les méthodes d'un traitement sont regroupées dans un objet visiteur
  - ⇒ facile de partager du code et des données communes
- Un objet visiteur contient pour chaque type  $\kappa$  d'arbre une méthode (appelée `case $\kappa$` ) qui peut traiter les arbres de ce type.
- L'arbre contient uniquement une méthode de traitement générique qui ne fait qu'appliquer un objet visiteur donné.

Martin Odersky, LAMP/DI

18

## Arbres visitables pour les expressions

(on a omis les constructeurs)

```
package calc;
abstract class Tree {
    int pos;
    abstract void apply(Visitor v);
    static class NumLit extends Tree {
        int value;
        void apply(Visitor v) { v.caseNumLit(this); }
    }
    static class Operation extends Tree {
        int operator; Tree left, right;
        void apply(Visitor v) { v.caseOperation(this); }
    }
}
interface Visitor {
    void caseNumLit(Tree.NumLit tree);
    void caseOperation(Tree.Operation tree);
}
```

Martin Odersky, LAMP/DI

19

## Un visiteur ToString

```
package calc;
import calc.Tree.*;
class ToString implements Visitor {
    String result;
    public void caseNumLit(NumLit tree) {
        result = String.valueOf(tree.value);
    }
    public void caseOperation(Operation tree) {
        result = "(" + visit(tree.left) +
            Scanner.representation(tree.operator) +
            visit(tree.right) + ")";
    }
    public static String visit(Tree tree) {
        ToString v = new ToString();
        tree.apply(v);
        return v.result;
    }
}
```

Martin Odersky, LAMP/DI

20

## Un visiteur qui évalue

```
package calc;
import calc.Tree.*;
class Eval implements Visitor {
    int result;
    public void caseNumLit(NumLit tree){
        result = tree.value;
    }
    public void caseOperation(
        Operation tree) {
        int l = visit(tree.left);
        int r = visit(tree.right);
        switch (tree.operator) {
        case Scanner.PLUS :
            result = l + r; break;
        case Scanner.MINUS:
            result = l - r; break;
        case Scanner.MUL:
            result = l * r; break;
        case Scanner.DIV:
            result = l / r; break;
        default:
            throw new InternalError();
        }
    }
    public static int visit(Tree tree){
        Eval v = new Eval();
        tree.apply(v);
        return v.result;
    }
}
```

Martin Odersky, LAMP/DI

21

## Classe pilote pour les visiteurs

```
package calc;
class Main {
    public static void main(String[] args) {
        System.out.print("> ");
        Tree t = new Parser(System.in).expression();
        if (t != null)
            System.out.println(ToString.visit(t) + " evaluates to "
                + Eval.visit(t));
    }
}
```

Martin Odersky, LAMP/DI

22

## Quelle est la meilleure solution ?

- Extensibilité
  - La décomposition OO facilite l'ajout de nouveaux types de nœud.
  - Les visiteurs facilitent l'ajout de nouveaux traitements.
- Modularité
  - La décomposition OO permet le partage de données et de code dans un nœud de l'arbre entre les phases.
  - Les visiteurs permettent le partage de données et de code entre les méthodes d'un même traitement.
- Qu'est-ce qui est le plus important ?

## Les arbres dans d'autres contextes

- Les arbres avec plusieurs types de nœuds n'interviennent pas que dans la compilation.
- On les retrouve aussi dans la mise en page de texte, dans les documents structurés tels que HTML ou XML, les interfaces graphiques utilisateur (*graphical user interface*).
- Exemple : Composants d'une GUI
  
- Quelle méthode d'accès à l'arbre est utilisée pour les composants d'une GUI ?
- Quel type d'extension est la plus commune ?

## Extensibilité

### Interprétations

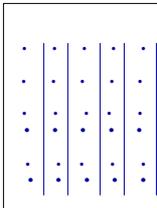
Vérification de typage  
 Traduire pour Pentium  
 Traduire pour Sparc  
 Trouver le code mort  
 Optimiser  
 . . . .

### Interprétations

Redessiner  
 Déplacer  
 Réduire  
 Maximiser  
 Activer  
 . . . .

Sortes

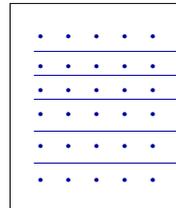
IdExp  
 NumExp  
 PlusExp  
 MinusExp  
 TimesExp  
 SeqExp



(a) Compilateur

Sortes

Scrollbar  
 Menu  
 Canvas  
 DialogBox  
 Text  
 StatusBar



(b) Interface graphiques utilisateur

Martin Odersky, LAMP/DI

25

## Syntaxe abstraite pour misc

<p>P = Program { D } B</p> <p>B = Body { S } E</p> <p>D = FunDef ident { F } T E</p> <p>F = Formal ident T</p> <p>T = BasicType U            ListType T            FunType { T } T</p> <p>U = Unit            Any            Int</p> <p>S = VarDef F E            While E E            Exec E</p>	<p>E = If E E E            Assign ident E            Binop O E E            Listop L E            Ident ident            UnitLit            IntLit int            NilLit            Apply E { E }            B</p> <p>O = Add   Sub   Mul   Div            Mod   Eq   Ne   Lt   Le            Gt   Ge   Cons</p> <p>L = Head   Tail   IsEmpty</p>
---	---

Martin Odersky, LAMP/DI

26

## De la syntaxe abstraite aux arbres de syntaxe abstraite

- Simplifier encore plus en fusionnant P, D, F, T, S et E

```
Tree = Program { Tree } Tree
      | FunDef String { Tree } Tree Tree
      | Body { Tree } Tree
      | Formal String Tree
      | BasicType int
      | ListType Tree
      | ...
```

- Définir une classe abstraite `Tree` avec des sous-classes internes concrètes `Program`, `FunDef`, `Formal`, `BasicType`, etc.
- Définir une interface visiteur avec les méthodes

```
caseProgram(Program tree)
caseFunDecl(FunDef tree)
...
```

Martin Odersky, LAMP/DI

27

## La classe `Tree` pour misc

```
package misc;

public abstract class Tree {
    // common for all trees
    int pos;

    abstract void apply(Visitor v);

    static class Program extends Tree {
        Tree[] funs;
        Tree body;

        Program(int pos,
                Tree[] funs, Tree body) {
            this.pos = pos;
            this.funs = funs;
            this.body = body;
        }

        public void apply(Visitor v) {
            v.caseProgram(this);
        }
    }

    static class Formal extends Tree {
        String name;
        Tree tpe;

        Formal(int pos,
              String name, Tree tpe) {
            this.pos = pos;
            this.name = name;
            this.tpe = tpe;
        }

        public void apply(Visitor v) {
            v.caseVarDecl(this);
        }
    }

    ...
}

public interface Visitor {
    void caseProgram(Tree.Program tree);
    void caseFormal(Tree.Formal tree);
    ...
}
```

Martin Odersky, LAMP/DI

28

## Explications

- Chaque classe a un constructeur pour construire un nœud du type donné et une méthode `apply` qui applique un visiteur.
- La répétition est exprimée par des tableaux. `{T}` dans la syntaxe devient `T[]` dans l'arbre.
- Les symboles terminaux sont représentés par leur information essentielle :
  - pour un `ident` : le nom de l'identificateur (`String`).
  - pour un nombre : sa valeur (`int`).
- Le champ `pos` contient la position courante dans l'arbre, importante pour les messages d'erreur. Ce champ est commun à tous les types d'arbres ; c'est pourquoi il est un membre de la classe `Tree`.

Martin Odersky, LAMP/DI

29

## Construction de l'arbre

```
package misc;

import java.util.List;
import java.util.LinkedList;
import misc.Tree.*;

class Parser extends Scanner {
    ...

    Tree formal() {
        int start = pos;
        String name = chars;
        accept(IDENT);
        accept(COLON);
        Tree type = type();
        return
            new Formal(start, name, type);
    }

    Tree program() {
        int start = pos;
        List list = new LinkedList();
        while (token == DEF) {
            list.add(declaration());
            accept(SEMICOLON);
        }
        Tree body = body();
        Tree[] funs =
            (Tree[])list.toArray(
                new Tree[list.size()]);
        accept(EOF);
        return
            new Program(start, funs, body);
    }
    ...
}
```

Martin Odersky, LAMP/DI

30

## Exemple de visiteur: un formateur d'arbre

```
package misc;

import misc.Tree.*;

public class Pretty implements Visitor {
    public void caseProgram(Program tree) {
        for (int i = 0; i < tree.funs; i++) {
            print(tree.funs[i]);
            System.out.println();
        }
        print(tree.body);
    }
    ...
    public void print(Tree tree) {
        tree.apply(this);
    }
}
```