Syntaxe abstraite

Martin Odersky

version 1.3

Plan du cours

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 :

analyseur à descente récursive : intégrées aux routines de reconnaissance.

analyseur ascendant généré automatiquement : ajoutées à la grammaire soumise au générateur d'analyseurs.

Exemple : Analyseur d'expressions arithmétiques

Exemple : Les expressions arithmétiques (E.A.)

```
E  =  E "+" T | E "-" T | T.
T  =  T "*" F | T "/" F | F.
F  =  number | "(" E ")".
```

Que l'on peut transformer en LL(1) :

```
E  =  T { "+" T | "-" T }.
T  =  F { "*" F | "/" F }.
F  =  number | "(" E ")".
```

Exemple : Analyseur d'E.A. (action sémantique)

```
class Parser(in:InputStream) extends Scanner(in)  {
  import Tokens._
  nextToken()
  def expression: Unit = {
    term
    while (token == PLUS || token == MINUS ) { nextToken(); term }
  }
  def term: Unit = {
    factor
    while (token == MUL || token == DIV) { nextToken(); factor }
  }
  def factor: Unit = {
    token match {
      case LITERAL => nextToken()
      case LPAREN =>
        nextToken()
        expression
        if (token == RPAREN) nextToken() else error("')' expected")
      case _ => error("illegal start of expression")
    }
}
```


Exemple : Analyseur d'E.A. (action sémantique : interprétation)

```
class Parser(in:InputStream) extends Scanner(in) {
  import Tokens._
  nextToken()
  def expression: Int = {
    var v: Int = term
    while (token == PLUS || token == MINUS ) {
      val operator = token; nextToken()
      v = if (operator == PLUS) v + term else v - term
    }
    v
  }
  def term: Int = {
    var v: Int = factor
    while (token == MUL || token == DIV) {
      val operator: Int = token; nextToken()
      v = if (operator == MUL) v * factor else v / factor
    }
    v
  }
  def factor: Int = {
    var v: Int = 0
    token match {
      case LITERAL => v = Integer.parseInt(chars); nextToken()
      case LPAREN => nextToken(); v = expression; accept(RPAREN)
      case _ => error("illegal start of expression")
    }
    v
}}
```


Arbres de syntaxe

Dans un compilateur multi-passes, l'analyseur construit explicitement un arbre de syntaxe.

- Les phases suivantes du compilateur travaillent sur l'arbre de syntaxe et non sur le source du programme.

Un arbre de syntaxe peut-être :

un arbre de syntaxe concrète s'il correspond directement à la grammaire non-contextuelle.

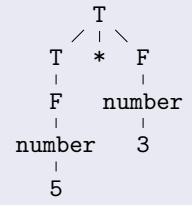
un arbre de syntaxe abstraite s'il correspond à une grammaire simplifiée.

Dans les compilateurs, on utilise généralement un arbre de syntaxe abstraite.

Arbres de syntaxe concrète

Exemple : Un arbre de syntaxe concrète pour une E.A.

En utilisant la grammaire non-contextuelle des expressions arithmétiques définies précédemment, on obtient pour l'expression `5 * 3` l'arbre de syntaxe concrète suivant :



Evidemment, un tel arbre contient une grande quantité d'information redondante.

Il est possible de simplifier l'arbre pour omettre cette information redondante :

- La structure en arbre permet de se passer de l'analyse du texte : les parenthèses ne sont pas nécessaires.

Exercice : Simplifier l'arbre de syntaxe concrète

Simplifiez `A * (B + C)`.

- Les symboles terminaux peuvent être implicites.

Simplifiez `if (x == 0) y = 1 else y = 2`.

- etc.

Grammaire abstraite

Ces simplifications vont amener à une nouvelle grammaire non-contextuelle plus simple : la grammaire abstraite.

Exemple : Grammaire abstraite pour les E.A.

```
Expr  =  BinOp Expr Op Expr
      |  IntLit int.
  Op  =  Add | Sub | Mul | Div.
```

Notez que pour une grammaire concrète donnée, il existe une multitude de grammaires abstraites possibles.

```
Expr  =  AddOp Expr Expr | SubOp Expr Expr
      |  MulOp Expr Expr | DivOp Expr Expr
      |  IntLit int.
```

On choisit la grammaire abstraite la plus adaptée au compilateur.

Arbres de syntaxe abstraite

Les arbres de syntaxe abstraite (*abstract syntax tree*, AST) constituent la structure de données centrale du compilateur.

On définit un type de noeud pour chaque alternative de chaque production de la syntaxe abstraite que l'on représente par un ensemble de classes :

- Une classe pour chaque alternative.
- Une super-classe abstraite commune : dans notre cas `Expr`.
- Les sous-arbres comme variables d'instance des classes.
- Un seul constructeur par classe.

En Scala, on utilise des classes `case` pour représenter les noeuds.

Exemple : Déclaration de l'AST pour les E.A.

```
abstract class Expr {
  var pos: Int
}

case class IntLit(value: Int)
     extends Expr

case class BinOp(op: Int, left: Expr, right: Expr)
     extends Expr
```

Exemple : Construction de l'AST pour les E.A.

```
class Parser(in:InputStream) extends Scanner(in) {
  nextToken()
  def expression: Expr = {
    var t: Expr = term
    while(token == PLUS || token == MINUS ) {
      val startpos: int = pos; val operator: int = token
      nextToken(); t = BinOp(operator, t, term)
    }
    t
  }
  def term: Expr = {
    var t: Expr = factor
    while (token == MUL || token == DIV) {
      val startpos: int = pos; val operator: int = token
      nextToken(); t = BinOp(operator, t, factor) setPos startPos
    }
    t
  }
  def factor: Expr = {
    var t: Expr = _
    token match {
      case LITERAL => t = IntLit(Integer.parseInt(chars)) setPos pos; nextToken()
      case LPAREN => nextToken(); t = expression; accept(RPAREN)
      case _ => error("illegal start of expression")
    }
    t
}}
```


Traitement des positions

- Il est important de stocker des positions dans l'AST pour pouvoir localiser les fautes qui sont détectées dans l'analyse des noms et types.
- Pour ca, on stocke un champ `pos` dans chaque noeud.
- Exemple :

```
abstract class Expr {
  var pos: Int = Position.UNDEFINED
  def setPos(p: Int): this.type = { pos = p; this }
}
```


- `pos` est commun à tous les types d'arbres ; c'est pourquoi il est membre de la classe `Tree`.
- `pos` ne fait pas partie du constructeur des classes, ceci afin de ne pas «polluer» le filtrage de motif :
  - `setPos` doit être appelée juste après la construction de la classe — c'est une sorte de post-constructeur.
  - `setPos` retourne `this` : il est ainsi possible de chaîner l'appel à `setPos` directement au constructeur :

```
val prog = Program(classes, main) setPos pos;
```


Accéder aux AST

Définition : Processeur

Conceptuellement, un processeur (*processor*) est une action en une seule traversée de l'AST

Dans un compilateur, un processeur est par exemple :

- une action qui donne à chaque noeud de l'arbre un type ;
- une optimisation ;
- la génération du code machine ;
- etc.

Il est important que les processeurs d'arbres puissent accéder à l'AST de manière flexible.

Comment les processeurs accèdent-ils à l'arbre ?

Une solution inefficace est d'utiliser `isInstanceOf` pour trouver le type du noeud syntaxique.

Exemple : Processeur grossier

```
if (tree.isInstanceOf[NumLit])
  tree.asInstanceOf[NumLit].value
```

Une bonne solution est la décomposition orientée-objet.

Une meilleure solution est la décomposition fonctionnelle (les «visiteurs»).

Nous allons maintenant présenter à la fois la décomposition orientée-objet et fonctionnelle en utilisant l'exemple des expressions arithmétiques.

On considère :

- Deux types de noeuds : `BinOp` et `IntLit`.
- Deux types d'actions : `eval` et `toString`.

La méthode s'applique de la même manière pour des langages comme Zwei (25 noeuds et 2 processeurs) ou Scala (44 noeuds et 16 processeurs).

Décomposition orientée-objet

Dans cette technique, chaque processeur d'arbre *P* est représenté par une méthode virtuelle `def P` dans chaque classe d'arbre.

- La méthode est abstraite dans la classe `Expr` et implémentée dans chaque sous-classe.
- Pour traiter une sous-expression `e`, il suffit d'appeler `e.P`, sa méthode correspondant au processeur.

Exemple : Décomposition orientée-objet sur les E.A.

Pour implanter notre exemple :

- on définit les méthodes `eval` et `toString` dans les classes `BinOp` et `IntLit` ;
- les méthodes `eval` et `toString` sont abstraites dans la classe `Expr`, donc elles peuvent être invoquées sur chaque arbre ;
- ce qu'elles font va dépendre du type concret de l'arbre.


```
abstract class Expr {
  var pos: Int
  def setPos(p: Int): this.type = { pos = p; this }
  override def toString: String
  def eval: Int
}
case class IntLit(value: int) extends Expr {
  def toString: String = value.toString
  def eval: Int = value
}
case class BinOp(op: int, left: Expr, right: Expr) extends Expr {
  def toString: String =
    "("+left.toString+Scanner.representation(op)+right.toString+")"
  def eval: Int = {
      val l: int = left.eval
      val r: int = right.eval
      op match {
        case Tokens.PLUS  => l + r
        case Tokens.MINUS => l - r
        case Tokens.MUL   => l * r
        case Tokens.DIV   => l / r
        case _            => throw new InternalError()
}}}
```

Exemple : Décomposition OO sur les E.A. (classe pilote)

```
object Main {
  def main(args: Array[String]): Unit = {
    Console.print(">␣")
    val t: Expr = new Parser(System.in).expression
    if (t != null) {
      Console.println(t.toString +
                      "␣evaluates␣to␣" +
                      t.eval)
    }
  }
}
```


Décomposition fonctionnelle à filtrage de motifs

Avec cette technique, toutes les méthodes d'un traitement sont regroupées dans un objet, le visiteur.

- On utilise le filtrage de motifs pour déterminer quel type de noeud est en train d'être traité.

Il est ainsi facile de partager du code et des données communes.

Exemple : Visiteur `toString` sur les E.A.

```
def toString(e: Expr): String = e match {
  case IntLit(value) => value.toString
  case BinOp(op, left, right) =>
    "(" + toString(left) + Scanner.representation(op) +
    toString(right) + ")"
}
```


Exemple : Visiteur `eval` sur les E.A.

```
def eval(e: Expr): Int = {
  e match {
    case IntLit(value)   =>
      value
    case BinOp(op, left, right) =>
      val l: Int = eval(left)
      val r: Int = eval(right)
      op match {
        case PLUS  => l + r
        case MINUS => l - r
        case MUL   => l * r
        case DIV   => l / r
        case _     => error("malformed␣op")
      }
  }
}
```


Exemple : Décomposition fonctionnelle sur les E.A. (classe pilote)

```
def main(args: Array[String]): Unit = {
  Console.print(">␣")
  val t: Expr = new Parser(System.in).expression
  if (t != null) {
    Console.println(toString(t)+"␣evaluates␣to␣"+eval(t))
  }
}
```

Extensibilité

Avec un arbre de syntaxe abstraite, il peut y avoir extension dans deux dimensions :

1. Ajouter un nouveau type de noeud, dans la décomposition :
   - orientée-objet ajouter une nouvelle sous-classe ;
   - fonctionnelle modifier chaque visiteur pour traiter le nouveau type de noeud.
2. Ajouter un nouveau type de méthode de traitement, dans la décomposition :
   - orientée-objet modifier chaque sous-classe pour y ajouter le traitement ;
   - fonctionnelle ajouter un nouvel objet visiteur.

Choisir le bon type de processeur

Faciliter l'extensibilité

- La décomposition OO facilite l'ajout de nouveaux types de noeud.
- Les visiteurs facilitent l'ajout de nouveaux traitements.

Faciliter la modularité

- La décomposition OO permet le partage des données et du code dans un noeud de l'arbre entre les phases.
- Les visiteurs permettent le partage des données et du code entre les méthodes d'un même traitement.

Les arbres dans d'autres contextes

Les arbres avec plusieurs types de noeuds n'interviennent pas que dans la compilation.

On les retrouve aussi :

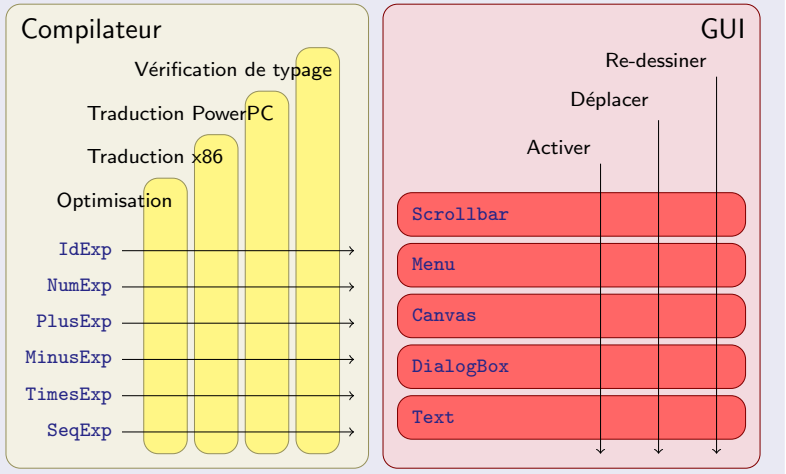
- dans la mise en page de texte ;
- dans les documents structurés tels que HTML ou XML ;
- dans les interfaces graphiques utilisateur (*GUI*) ;
- etc.

Exercice : Composants d'une GUI

Dans une interface graphique utilisateur :

- Quelle méthode d'accès à l'arbre est utilisée ?
- Quel type d'extension est la plus commune ?

Exemple : Extensibilité dans un compilateur et une GUI



Le motif de conception «visiteur»

Les langages ne supportant pas le filtrage de motif (comme Java) rendent la décomposition fonctionnelle plus difficile.

Le motif de conception (*design pattern*) «visiteur» permet de contourner cette difficulté :

- Un objet visiteur contient pour chaque type K d'arbre une méthode (appelée `caseK`) 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.

Référence : Motif de conception «visiteur»

Design Patterns, Elements of Reusable OO Software, E. Gamma, R. Helm, R. Johnson et J. Vlissides ; Addison-Wesley, 2000.

Exemple Java : Arbre visitable pour les E.A.

(Les constructeurs ont été omis)

```
public abstract class Expr {
    int pos;
    public abstract void apply(Visitor v);
    static class IntLit extends Expr {
        int value;
        public void apply(Visitor v) { v.caseIntLit(this); }
    }
    static class BinOp extends Expr {
        int operator;
        Expr left, right;
        public void apply(Visitor v) { v.caseBinOp(this); }
    }
    public interface Visitor {
        public void caseIntLit(IntLit tree);
        public void caseBinOp(BinOp tree);
    }
}
```


Exemple Java : Un visiteur `ToString` pour les E.A.

```
public class ToString implements Expr.Visitor {
    String result;
    public void caseIntLit(IntLit expr) {
        result = String.valueOf(expr.value);
    }
    public void caseBinOp(BinOp expr) {
        result = "("
               + visit(expr.left)
               + Scanner.representation(expr.operator)
               + visit(expr.right) + ")";
    }
    public static String visit(Expr tree) {
        ToString v = new ToString();
        tree.apply(v);
        return v.result;
    }
}
```


Exemple Java : Un visiteur `Eval` pour les E.A.

```
public class Eval implements Expr.Visitor {
    int result;
    public void caseIntLit(IntLit expr) {
        result = expr.value;
    }
    public void caseBinOp(BinOp expr) {
        int l = visit(expr.left);
        int r = visit(expr.right);
        switch (expr.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(Expr tree) {
        Eval v = new Eval();
        tree.apply(v);
        return v.result;
    }
}
```

Exemple Java : Décomposition par visiteur (classe pilote)

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