

Informatique 07

Analyse syntaxique

Expressions arithmétiques

François Bourdoncle

`Francois.Bourdoncle@ensmp.fr`

`http://www.ensmp.fr/~bourdonc/`

Analyse syntaxique : définition

- Reconnaissance d'une *phrase* d'un langage :
 - Expressions arithmétiques : évaluation, calcul symbolique, etc.
 - Langages de programmation : interprétation, compilation, mise en page, etc.
 - Langage naturel : vérification, traduction, etc.
- *Mots* du langage (analyse *lexicale*)

Variable = $a \mid \dots \mid z$

Chiffre = $0 \mid \dots \mid 9$

- *Syntaxe* du langage (*grammaire*)

Expr = Prod + Expr | Prod – Expr | Prod

Prod = Fact * Prod | Fact / Prod | Fact

Fact = Variable | Chiffre | (Expr)

- Analyse syntaxique :

syntaxe concrète \Rightarrow syntaxe abstraite

Expressions arithmétiques : définition

```
abstract class Expr {
    abstract void print();
}

class Chiffre extends Expr {
    int valeur;

    Chiffre(int valeur) { this.valeur = valeur; }
    void print() { write(valeur); }
}

class Variable extends Expr {
    char nom;

    Variable(char nom) { this.nom = nom; }
    void print() { write(nom); }
}

class Operateur extends Expr {
    char op;
    Expr e1;
    Expr e2;

    Operateur(char op, Expr e1, Expr e2) {
        this.op = op;
        this.e1 = e1;
        this.e2 = e2;
    }
    void print() {
        write("("); e1.print(); write(op);
        e2.print(); write(")");
    }
}
```

Expressions arithmétiques : utilisation

```
class Expr {
    ...
    public static void main(char[] args) {
        Expr e =
            new Operateur('-',
                new Variable('x'),
                new Operateur('+',
                    new Variable('y'),
                    new Chiffre(4)));
        e.print();
    }
    ...
}
```

```
%java Expr
x-(y+4)
```

Analyse syntaxique par descente récursive

```
class Expr {
    // Mot courant
    static char mot;

    // Analyse lexicale
    static void motSuivant() {
        do {
            // Caractère suivant
            int c = System.in.read();
            if (c == -1) {
                erreur("fin de fichier");
            } else {
                mot = (char) c;
            }
        } while (Character.isSpaceChar(mot));
    }

    // Analyse syntaxique
    static Expr lirePhrase() { ... }
    static Expr lireFact() { ... }
    static Expr lireProd() { ... }
    static Expr lireExpr() { ... }
}
```

Lecture d'une phrase

```
static Expr lirePhrase() {  
    // Lire le premier mot  
    motSuivant();  
  
    // Lire l'expression  
    Expr e = lireExpr();  
  
    // Vérifier que le mot suivant est ';'   
    if (mot != ';') {  
        erreur("Point-virgule manquant");  
    }  
  
    // Retourner l'expression lue  
    return e;  
}
```

Lecture d'un facteur

```
static Expr lireFact() {
    if ('a' <= mot && mot <= 'z') {
        Expr e = new Variable(mot);
        motSuivant();
        return e;
    } else if ('0' <= mot && mot <= '9') {
        Expr e = new Chiffre((int) mot - (int) '0');
        motSuivant();
        return e;
    } else if (mot == '(') {
        motSuivant();
        Expr e = lireExpr();
        if (mot != ')') {
            erreur("Parenthèse manquante");
        }
        motSuivant();
        return e;
    } else {
        erreur("Erreur de syntaxe");
    }
}
```

Lecture d'un produit de facteurs

```
static Expr lireProd() {  
    Expr e1 = lireFact();  
    if (mot == '*' || mot == '/') {  
        char op = mot;  
        motSuivant();  
        Expr e2 = lireProd();  
        return new Operateur(op, e1, e2);  
    } else {  
        return e1;  
    }  
}
```

Lecture d'un produit de facteurs (version correcte)

```
static Expr lireProd() {
    Expr e1 = lireFact();
    while (mot == '*' || mot == '/') {
        char op = mot;
        motSuivant();
        Expr e2 = lireFact();
        e1 = new Operateur(op, e1, e2);
    }
    return e1;
}
```

Lecture d'une expression

```
static Expr lireExpr() {
    Expr e1 = lireProd();
    if (mot == '+' || mot == '-') {
        char op = mot;
        motSuivant();
        Expr e2 = lireExpr();
        return new Operateur(op, e1, e2);
    } else {
        return e1;
    }
}
```

Lecture d'une expression (version correcte)

```
static Expr lireExpr() {  
    Expr e1 = lireProd();  
    while (mot == '+' || mot == '-') {  
        char op = mot;  
        motSuivant();  
        Expr e2 = lireProd();  
        e1 = new Operateur(op, e1, e2);  
    }  
    return e1;  
}
```

Analyse syntaxique : résumé

- Analyse lexicale : séparation en mots

```
class Mot {  
    // Les caracteres du mot  
    String caracteres;  
  
    // La catégorie du mot  
    int categorie;  
  
    // Les diverse catégories de mots  
    static final int NOMBRE = 0;  
    static final int IDENT = 1;  
    static final int PUBLIC = 2;  
    static final int STATIC = 3;  
    static final int LPAR = 4;  
    static final int RPAR = 5;  
    ...  
}  
  
class Expr {  
    static Mot mot;  
    static void motSuivant() { ... }  
}
```

- Calculer l'ensemble des mots possibles à chaque alternative de la grammaire
 - Le programme est ensuite mécaniquement calqué sur la grammaire
 - Attention aux grammaires récursives à gauche
- $\text{Exp} = \text{Exp} + \text{Prod} \mid \text{Exp} - \text{Prod} \mid \text{Prod}$
- Générateur d'analyseur syntaxique (YACC)

Exercices

- Impression postfixe : $x \ y \ z \ * \ +$
- Impression préfixe : $(+ \ x \ (* \ y \ z))$
- Impression infixe : $x + y * z$
- Evaluation étant donnée la valeur des variables (interpréteur)
- Simplification en utilisant les règles :

$$0 + x \longrightarrow x$$

$$x + 0 \longrightarrow x$$

$$1 * x \longrightarrow x$$

$$x * 1 \longrightarrow x$$

$$0 * x \longrightarrow 0$$

- Dérivation formelle