

Cours 5

Analyse syntaxique

Jean-Jacques.Levy@inria.fr

<http://jeanjacqueslevy.net>

tel: 01 39 63 56 89

secrétariat de l'enseignement:

Catherine Bensoussan

cb@lix.polytechnique.fr

Aile 00, LIX

tel: 01 69 33 34 67

<http://www.enseignement.polytechnique.fr/informatique/>

Plan

1. Analyse lexicale
2. Grammaires formelles
3. BNF
4. Syntaxe abstraite
5. Analyse récursive descendante

Analyse lexicale

```
class Lexeme {
    final static int L_nombre=0, L_id=1, L_Plus='+', L_Moins='-',
        L_Mul='*', L_Div='/', L_ParO='(', L_ParF=')',
        L_EOF = -1;
    int nature; String nom; int val;

    Lexeme (int i) { nature = L_nombre; val = i; }
    Lexeme (String s) { nature = L_id; nom = s; }
    Lexeme (char op) { nature = op; as_op = op}

    static char c;

    static Lexeme next () {
        skipBlanks();
        if (isLetter(c)) return new Lexeme (nextIdent());
        else if (isDigit(c)) return new Lexeme (nextNumber());
        else switch (c) {
            case '+': case '-': case '*': case '/':
            case '(': case ')': nextChar(); return new Lexeme (c);
            default: throw new Error ("Caractère illégal");
        }
    }
}
```

```
}  
  
static void skipBlanks() { while isWhitespace (c) nextChar(); }  
static int nextChar() { c = in.readChar(); }  
  
static String nextIdent() {  
    StringBuffer r;  
    while (isLetterOrDigit (c)) {  
        r = r.append (c);  
        nextChar();  
    }  
    return r;  
}  
  
static int nextIdent() {  
    int r = 0;  
    while (isDigit (c)) {  
        r = r * 10 + c - '0';  
        nextChar();  
    }  
    return r;  
}
```

Représentation des termes

On veut faire du calcul symbolique sur des termes. On représente les termes par des arbres. Par exemple:

```
class Terme {  
    final static int ADD = 0, SUB = 1, MUL = 2, DIV = 3, MINUS = 4,  
                  VAR = 5, CONST = 6;  
    int nature;  
    int valeur; String nom;  
    Terme a1, a2, a3;  
  
    Terme (int t, Terme a) {nature = t; a1 = a; }  
    Terme (int t, Terme a, Terme b) {nature = t; a1 = a; a2 = b; }  
    Terme (String s) {nature = VAR; nom = s; }  
    Terme (int v) {nature = CONST; valeur = v; }  
}
```

Surcharge

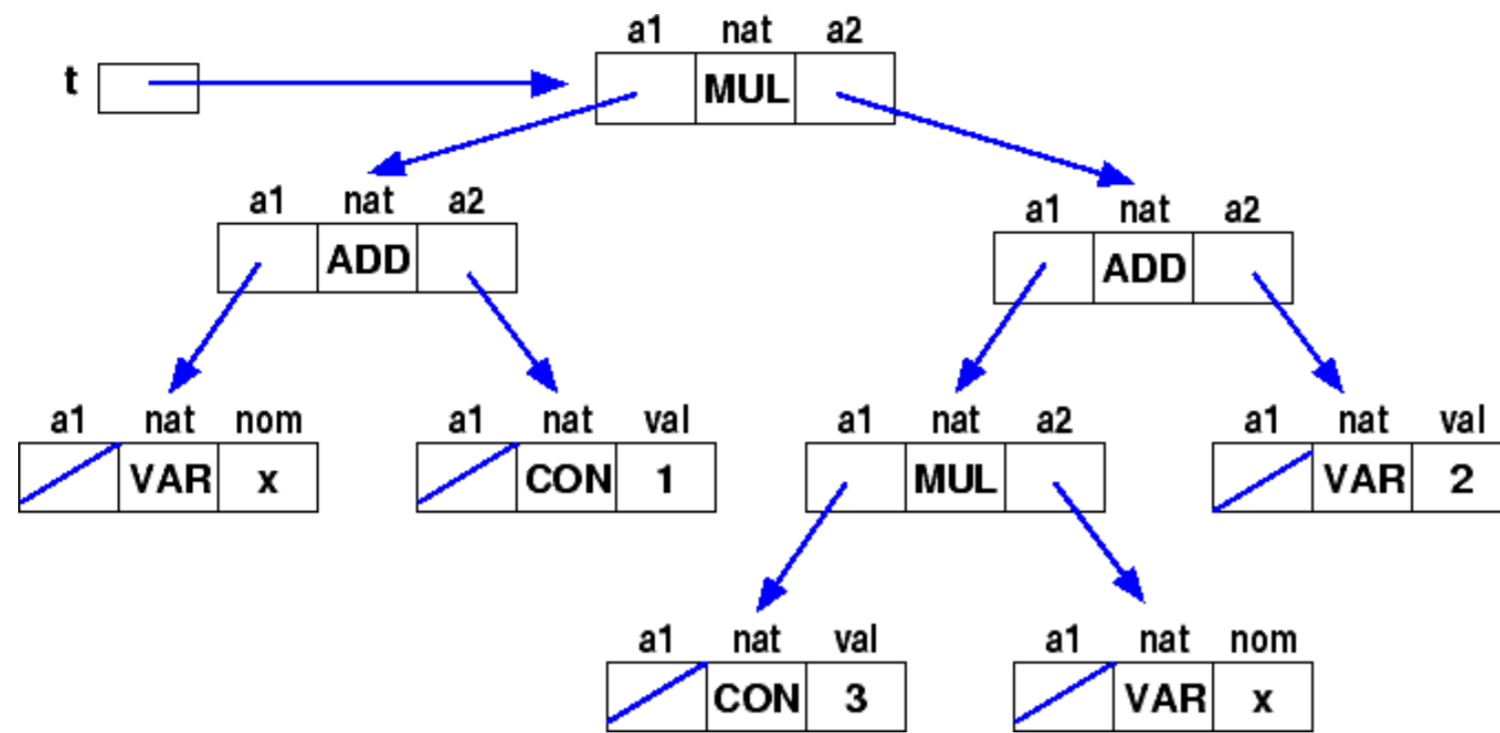
Les fonctions peuvent être **surchargées**, et donc aussi les constructeurs. Dans un constructeur, on peut appeler un constructeur de la même classe par `this()`.

```
class Terme {  
    final static int ADD = 0, SUB = 1, MUL = 2, DIV = 3, MINUS = 4,  
                  VAR = 5, CONST = 6;  
  
    int nature;  
    int valeur; String nom;  
    Terme a1, a2, a3;  
  
    Terme (int t, Terme a) {nature = t; a1 = a; }  
    Terme (int t, Terme a, Terme b) {nature = t; a1 = a; a2 = b; }  
    Terme (String s) { this(VAR, null, null) ; nom = s; }  
    Terme (int v) { this(CONST, null) ; valeur = v; }
```

Surcharge et polymorphisme sont deux notions différentes.

Exemples de termes

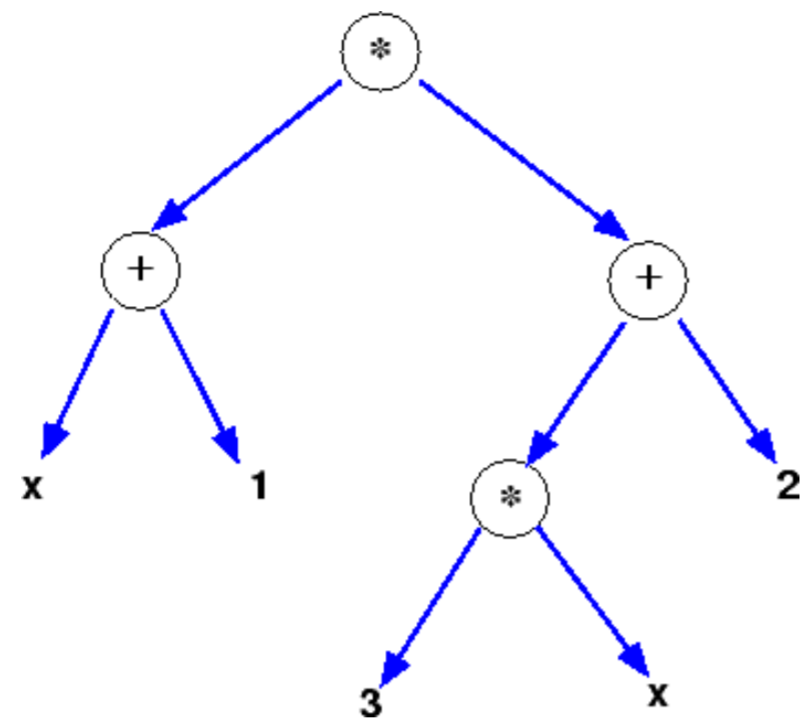
```
Terme t = new Terme (MUL,  
    new Terme (ADD, new Terme ("x"), new Terme (1)),  
    new Terme (ADD,  
        new Terme (MUL, new Terme (3), new Terme ("x")),  
        new Terme (2)));
```



On a choisi ici de faire l'union de tous les champs. D'autres représentations possibles seront vues plus tard.

Représentation des termes

ou encore pour $(x + 1) * (3 * x + 2)$



Arbres de syntaxe abstraite (*Abstract Syntax Tree*) ou **ASA** (*AST*).

Exercice 1 Dessiner les ASA pour $x + y + z$, $x * y * z$, $x * y + z$, $x * (y + z)$, $(a + a * a) * (a * a + a * a)$.

Grammaires formelles

On peut montrer que les automates finis sont insuffisants pour reconnaître des langages de la forme $\{a^n b^n \mid n \geq 0\}$.

On fait appel aux grammaires **algébriques** (*context-free*).

$G = (\Sigma, V, S, \mathcal{P})$ où

Σ alphabet **terminal**

V ensemble fini des variables **non terminales** ($V \cap \Sigma = \emptyset$)

S **axiome** ($S \in V$)

\mathcal{P} ensemble fini de règles de productions $X_i \rightarrow w_i$ ($X_i \in V, w_i \in (\Sigma \cup V)^*$)

Langage reconnu par G

- $v \rightarrow v'$ implique $uvw \rightarrow uv'w$
- $L(G) = \{w \mid S \rightarrow^* w, w \in \Sigma^*\}$

Les grammaires formelles sont enseignées dans les cours Automates et Calculabilité en majeure M1; les analyseurs syntaxiques en Compilation en majeure M2.

Exercice 2 Trouver les langages reconnus par les grammaires suivantes

Langage parenthésé

$\Sigma = \{a, b\}, V = \{S\}$

$S \rightarrow SS$

$S \rightarrow aSb$

$S \rightarrow$

Représentation linéaire d'arbres (cf. cours 3)

$\Sigma = \{[,], nb\}, V = \{A\}$

$A \rightarrow [A nb A]$

$A \rightarrow$

Expressions arithmétiques

$\Sigma = \{ (,), +, -, *, /, id, nb \}, V = \{ E, P, F \}, (E \text{ axiome})$

$E \rightarrow P + E \quad E \rightarrow P - E \quad E \rightarrow P$

$P \rightarrow F * P \quad P \rightarrow F / P \quad P \rightarrow F$

$F \rightarrow id \quad F \rightarrow nb \quad F \rightarrow (E)$

Syntaxe BNF (*Backus-Naur Form*)

La syntaxe des langages de programmation est aussi décrite par une grammaire formelle. Les nombreuses variables non-terminales sont décrites par des identificateurs. Il y a aussi des raccourcis pour simplifier la notation. Exemple (cf. cours 1):

ForStatement:

```
for ( ForInitopt ; Expressionopt ; ForUpdateopt )  
    Statement
```

ForInit:

```
StatementExpressionList  
LocalVariableDeclaration
```

ForUpdate:

```
StatementExpressionList
```

StatementExpressionList:

```
StatementExpression  
StatementExpressionList , StatementExpression
```

StatementExpression:

Assignment

PreIncrementExpression

PreDecrementExpression

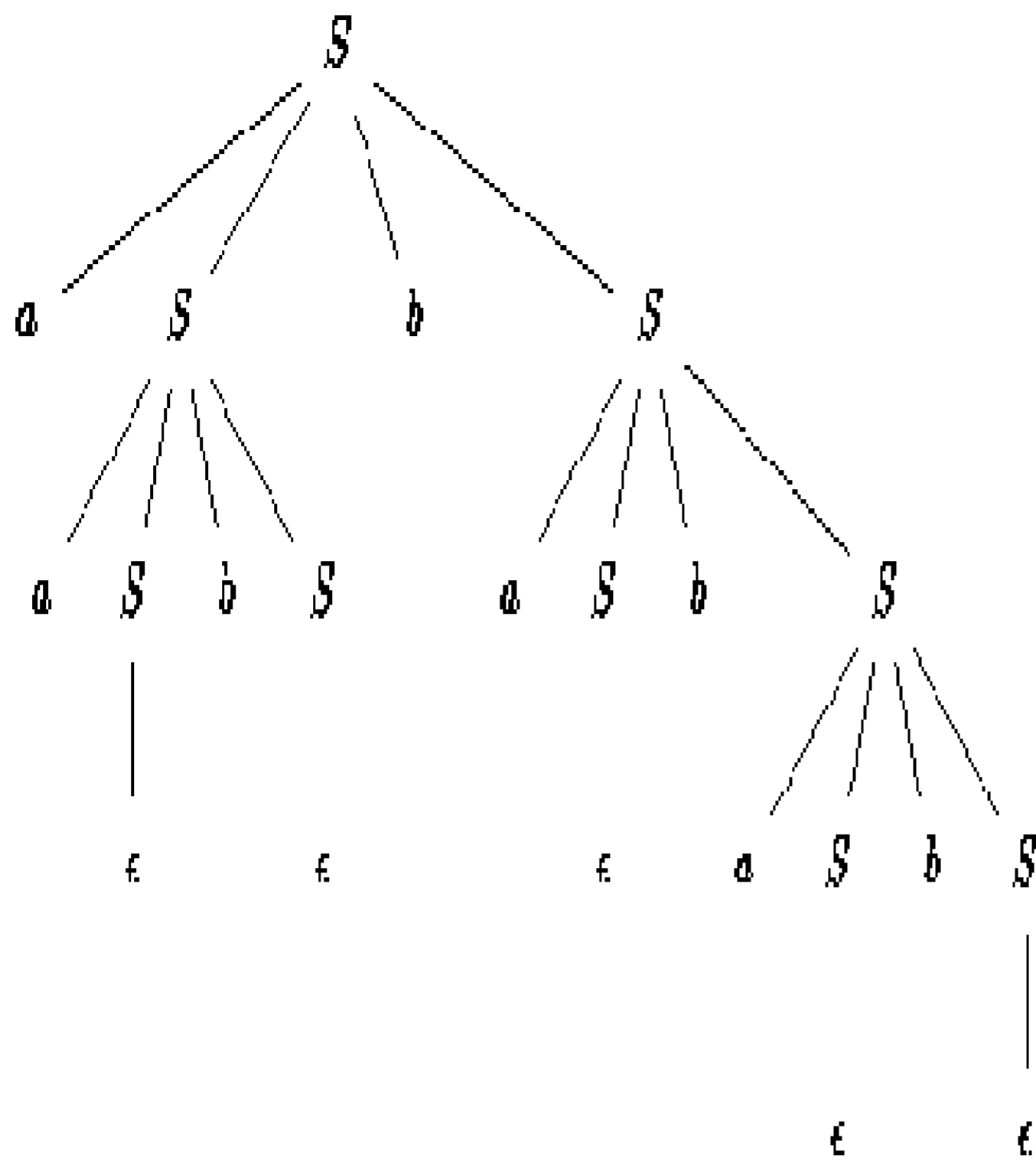
PostIncrementExpression

PostDecrementExpression

MethodInvocation

ClassInstanceCreationExpression

Arbre de dérivation (*Parse Tree*)

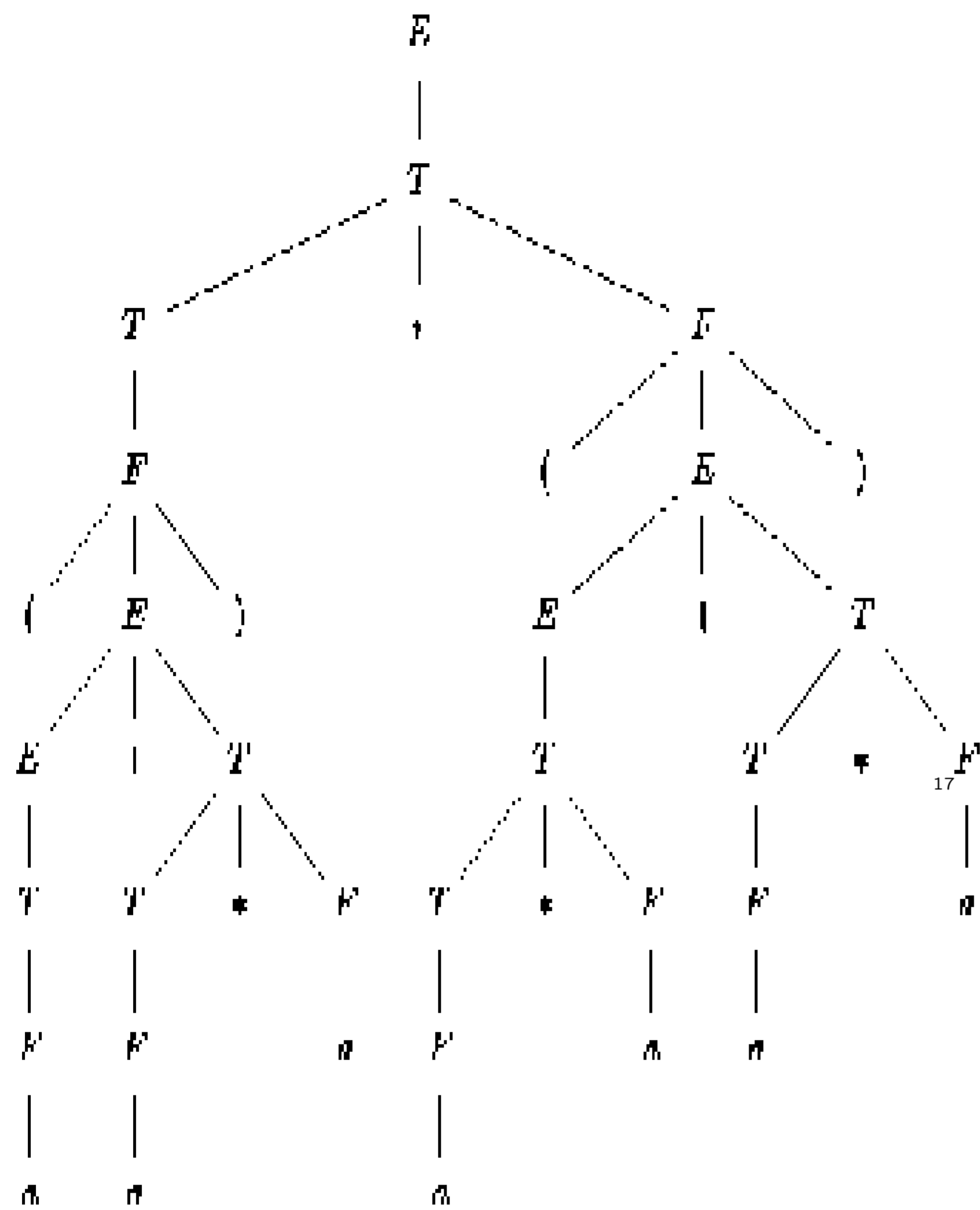


pour *aabbabab*

Si plusieurs arbres de dérivations sont possibles pour un même mot w , la grammaire est dite **ambigue**.

Exercice 3 Parmi les grammaires précédentes, lesquelles sont ambiguës?

Arbre de dérivation (Parse Tree) de $(a + a * a) * (a * a + a * a)$



Analyse syntaxique

Donnés: grammaire G et un mot w

But: $w \in L(G)$? Si oui, construire l'ASA de w .

2 méthodes

- **analyse descendante**. On part de S pour atteindre w .
(méthode de javacc)
grammaires $LL(k)$
- **analyse ascendante**. On part de w pour atteindre S .
(méthode de la commande yacc [S. Johnson] d'Unix)
grammaires $LR(k)$

Méthode récursive descendante (*Recursive descent*)

Le lexème courant dans une variable globale `curToken`.

`Lexeme.next()` va chercher le suivant.

Une procédure (récursive) par variables non terminale.

Au début on appelle la fonction correspondant à l'axiome.

```
static Lexeme curToken;
static void avancer() {curToken = Lexeme.next(); }

static Terme expression() {
    Terme t = produit(); switch (curToken.nature) {
        case Lexeme.L_Plus: avancer(); return new Terme (ADD, t, expression());
        case Lexeme.L_Moins: avancer(); return new Terme (MINUS, t, expression());
        default: return t;
    }
}
```

Méthode récursive descendante

```
static Terme produit() {
    Terme t = facteur(); switch (curToken.nature) {
        case Lexeme.L_Mul: avancer(); return new Terme (MUL, t, expression());
        case Lexeme.L_Div: avancer(); return new Terme (DIV, t, expression());
        default: return t;
    }
}

static Terme facteur() {
    Terme t; switch (curToken.nature) {
        case Lexeme.L_ParO: avancer(); t = expression();
            if (curToken.nature != Lexeme.L_ParF) throw new Error ("Il manque ')'");
            break;
        case Lexeme.L_nombre: t = new Terme (curToken.val); break;
        case Lexeme.L_id: t = new Terme (curToken.nom); break;
        default: throw new Error ("Erreur de syntaxe");
    }
    avancer();
    return t;
}
```

On décide toujours avec pas plus d'un caractère d'avance *LL(1)*.

Opérations sur syntaxe abstraite

- **belle impression** (*pretty-print*), i.e. sans parenthèses superflues.
- **interprétation** (évaluation d'expressions arithmétiques ou booléennes)
- calcul formel (dérivation formelle, intégration, etc.)
- **compilation** (génération de code)
- transformations (passer en notation polonaise postfixe ou préfixe).
- **analyses statiques**

Evaluation d'expressions arithmétiques

Données: t terme, e liste d'association de valeurs aux variables.

But: calcul la valeur du terme t dans l'environnement e .

```
static int evaluer (Terme t) {
    switch (t.nature) {
        case PLUS: return evaluer (t.a1) + evaluer (t.a2);
        case MINUS: return evaluer (t.a1) - evaluer (t.a2);
        case MUL: return evaluer (t.a1) * evaluer (t.a2);
        case DIV: return evaluer (t.a1) / evaluer (t.a2);
        case NOMBRE: return t.val;
        case VAR: return assoc (t.nom, e);
        default: throw new Error ("Erreur dans evaluation");
    }
}

static int assoc (String s, Environnement e) {
    if (e == null) throw new Error ("Variable non définie");
    if (e.nom.equals(s)) return e.val;
    else return assoc (s, e.suivant);
}
```

Exercice 4 Programmer la belle impression.

Exercice 5 Programmer la dérivation formelle.

Exercice 6 Essayer d'imaginer ce que pourrait être l'analyse ascendante.

Exercice 7 Trouver une solution pour l'analyse syntaxique des opérateurs associatifs.

Exercice 8 Donner des exemples où l'analyse descendante a des difficultés, mais pas une analyse ascendante.

En TD

- faire une calculatrice logique
- HP → TI