

Inf 431 – Cours 6

# Analyse syntaxique

[jeanjacqueslevy.net](http://jeanjacqueslevy.net)

secrétariat de l'enseignement:

Catherine Bensoussan

[cb@lix.polytechnique.fr](mailto:cb@lix.polytechnique.fr)

Aile 00, LIX,

01 69 33 34 67

[www.enseignement.polytechnique.fr/informatique/IF](http://www.enseignement.polytechnique.fr/informatique/IF)

# Plan

1. Arbres de syntaxe abstraite (arbres binaires, termes)
2. Grammaires formelles
3. BNF
4. Arbres syntaxiques
5. Analyse récursive descendante
6. Evaluation d'expressions arithmétiques
7. Associativité

cf. les cours Automates et Calculabilité en majeure M1 pour les grammaires formelles, et Compilation en majeure M2 pour les analyseurs syntaxiques

# Schéma général

Texte



Analyse lexicale



Flot de lexèmes



Analyse syntaxique



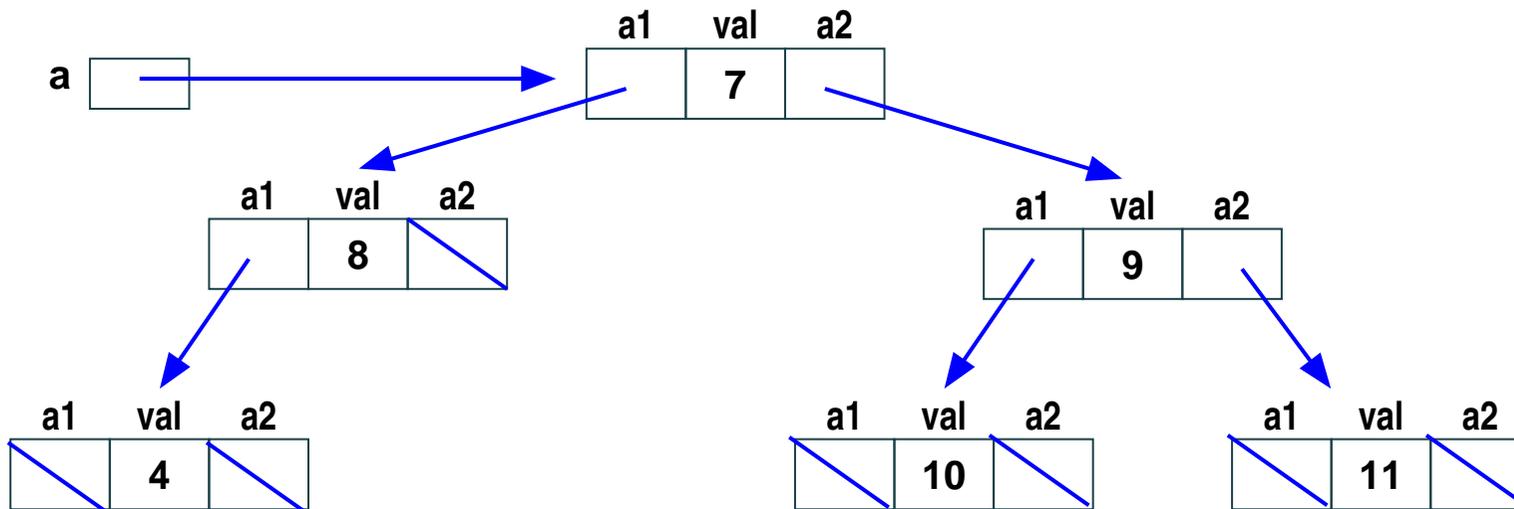
Arbre Syntaxe Abstraite (ASA)

# Arbres de syntaxe abstraite

- Le texte est **non structuré**
- ASA = **structure** de donnée abstraite pour traiter un problème
- Exemples de problèmes :
  - lire/écrire des **arbres binaires**
  - calculer des **expressions arithmétiques**
  - interpréter un petit langage de **commandes**
  - calculer **symboliquement** (algèbre, logique)
  - générer du code machine (**compilation**)
- Exemples d'ASAs :
  - arbres binaires
  - termes représentant une expression arithmétique

# Arbres binaires (1/2)

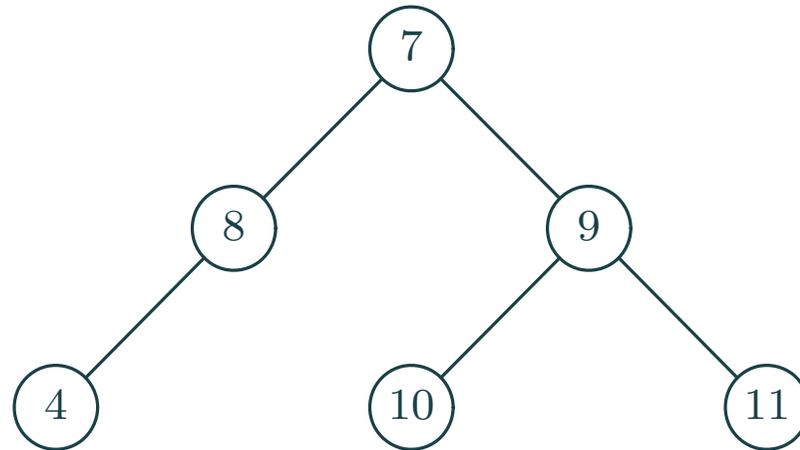
```
class Arbre {  
    int val;  
    Arbre a1, a2;  
  
    Arbre (int v, Arbre a, Arbre b) {val = v; a1 = a; a2 = b; }  
    Arbre (int v) {val = v; a1 = null; a2 = null; }  
}
```



```
Arbre a = new Arbre (7,  
    new Arbre (8, new Arbre(4), null),  
    new Arbre (9, new Arbre(10), new Arbre (11)));
```

## Arbres binaires (2/2)

ou encore plus abstraitement



# Surcharge – Overloading

- Les fonctions (et les constructeurs) peuvent être **surchargées**.
- Surcharge (Java)  $\neq$  **polymorphisme** (ML)
  - Une fonction polymorphe est **la même** pour une collection de types de ses arguments. Exemple : l'image miroir d'une liste (sans préciser le type de ses éléments).
  - Une fonction surchargée **change de sens** selon le type de ses arguments.
- Les deux notions sont **statiques**. Elles n'interviennent qu'à la phase de compilation (et non à l'exécution).

Les principes des langages de programmation modernes sont enseignés en majeure 1.

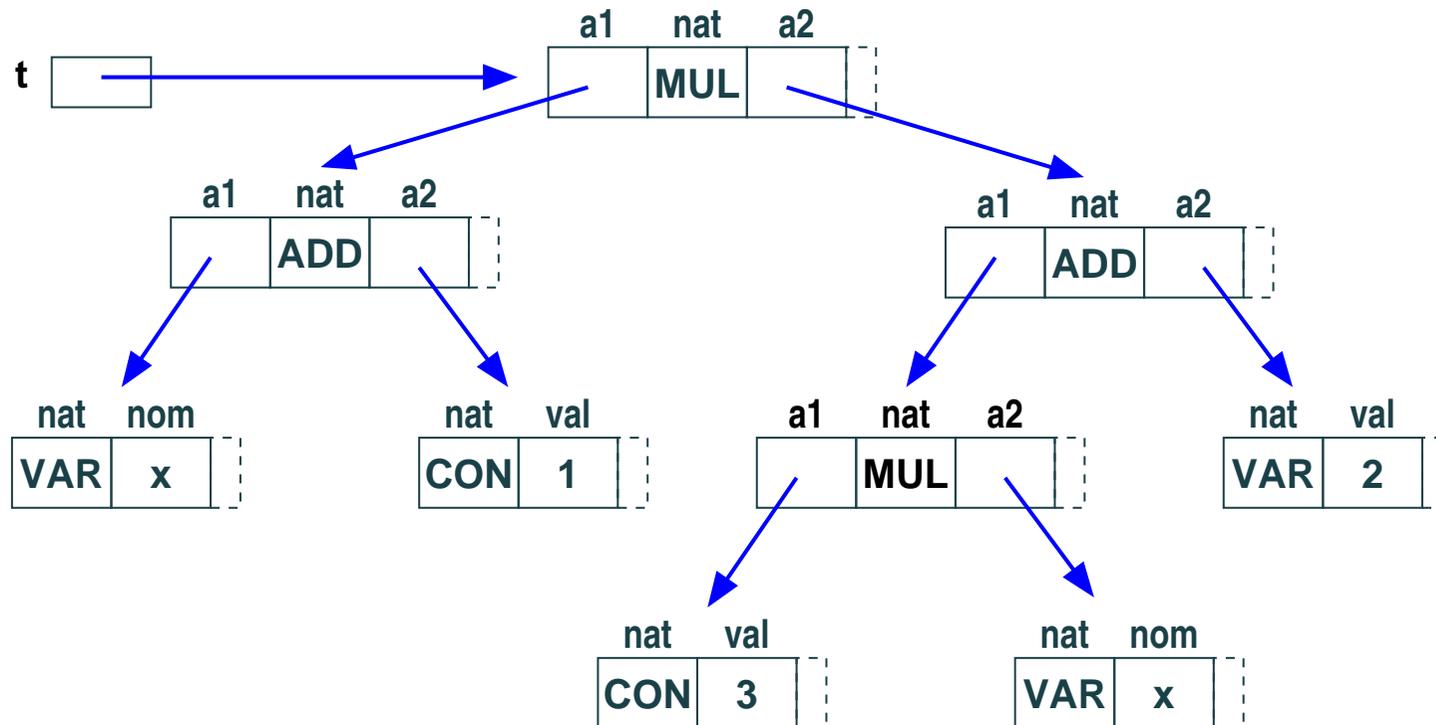
# Termes (1/3)

But : calculer. Les termes sont représentés 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;  
    Terme a1, a2;  
    String nom;  
    int val;  
  
    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; val = v; }  
}
```

# Termes (2/3)

```
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)));
```

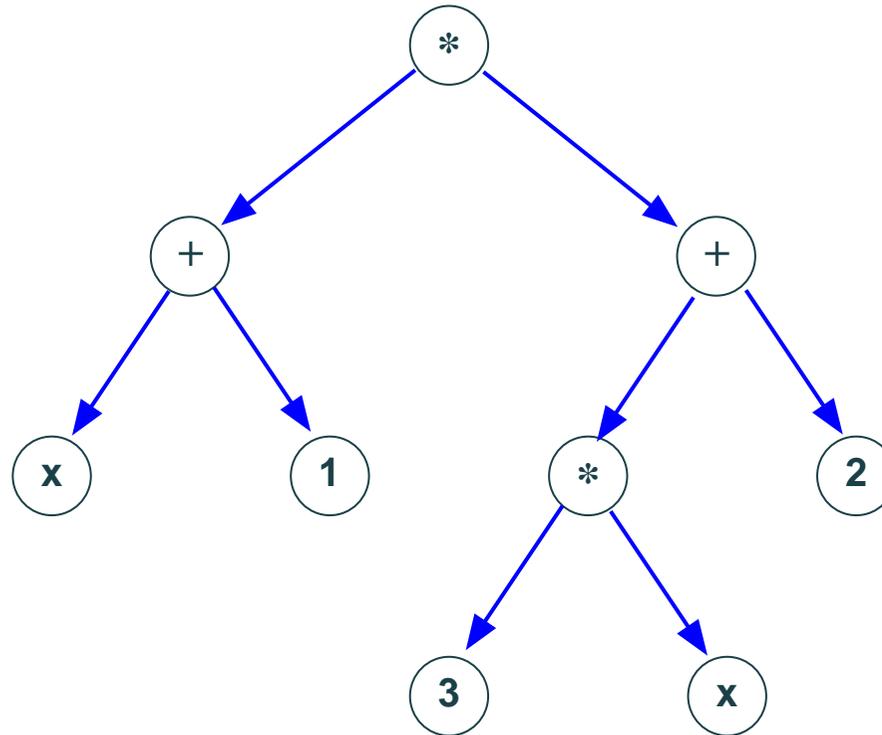


On a choisi ici de faire l'union de tous les champs. Seuls les plus significatifs figurent sur cette figure.

D'autres représentations seront vues plus tard.

# Termes (3/3)

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



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

# Grammaires formelles (1/4)

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

$G = (\Sigma, V, S, \mathcal{P})$  où

$\Sigma$  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  $uvw \rightarrow uv'w$
- $u \rightarrow^* v$  ssi  $u = u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \dots u_n = v$  ( $n \geq 0$ )
- $L(G) = \{w \mid S \rightarrow^* w, w \in \Sigma^*\}$

# Grammaires formelles (2/4)

## Langage parenthésé

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

$$S \rightarrow SS$$

$$S \rightarrow aSb$$

$$S \rightarrow$$

## Représentation linéaire d'arbres

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

$$A \rightarrow [ A nb A ]$$

$$A \rightarrow$$

## Expressions arithmétiques 1

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

$$E \rightarrow E + E$$

$$E \rightarrow E - E$$

$$E \rightarrow E * E$$

$$E \rightarrow E / E$$

$$E \rightarrow id$$

$$E \rightarrow nb$$

$$E \rightarrow ( E )$$

## Expressions arithmétiques 2

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

$$E \rightarrow P + E$$

$$E \rightarrow P - E$$

$$E \rightarrow P$$

$$P \rightarrow F * P$$

$$P \rightarrow F / P$$

$$P \rightarrow F$$

$$F \rightarrow id$$

$$F \rightarrow nb$$

$$F \rightarrow ( E )$$

# Grammaires formelles (3/4)

## Langage parenthésé

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

$$S \rightarrow SS$$

$$S \rightarrow aSb$$

$$S \rightarrow$$

$$S \rightarrow SS \rightarrow SaSb \rightarrow SaSSb \rightarrow SaSb \rightarrow SaaSbb \rightarrow Saabb \rightarrow aSbaabb \rightarrow abaabb$$

$$S \rightarrow aSb \rightarrow aSSb \rightarrow aSSSb \rightarrow aSaSbSb \rightarrow aSaSSbSb \rightarrow aaSbaSSbSb \rightarrow aaaSbbaSSbSb \\ \rightarrow aaaSbbaSSbaSbb \rightarrow aaabbaSSbaSbb \rightarrow aaabbaSbaSbb \rightarrow aaabbabaSbb \rightarrow aaabbababb$$

$a$  correspond à parenthèse ouvrante '(' ,

$b$  correspond à parenthèse fermante ') ' .

# Grammaires formelles (4/4)

Représentation linéaire des arbres binaires

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

$$A \rightarrow [ A nb A ]$$

$$A \rightarrow$$

$$\begin{aligned} A \rightarrow [ A7A ] \rightarrow [[ A8A ] 7A ] \rightarrow [[[ A4A ] 8A ] 7A ] \rightarrow [[ [ [ 4A ] 8A ] 7A ] \\ \rightarrow [ [ [ [ 4 ] 8A ] 7A ] \rightarrow [ [ [ [ 4 ] 8 ] 7A ] \rightarrow [ [ [ [ 4 ] 8 ] 7 [ A9A ] ] \\ \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ A10A ] 9A ] ] \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ 10A ] 9A ] ] \\ \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9A ] ] \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ A11A ] ] ] \\ \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11A ] ] ] \rightarrow [ [ [ [ 4 ] 8 ] 7 [ [ 10 ] 9 [ 11 ] ] ] \end{aligned}$$

Long mais **précis**!

On peut abstraire les dérivations par un arbre de dérivations ou encore **arbre syntaxique**.

# Arbre syntaxique (1/5)

mot

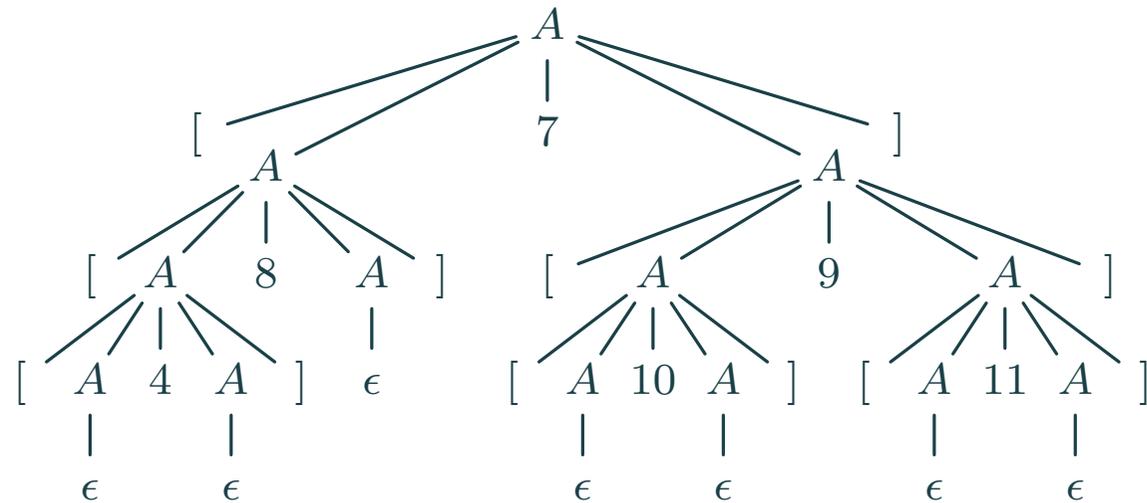
[[[4]8]7[[10]9[11]]]

grammaire

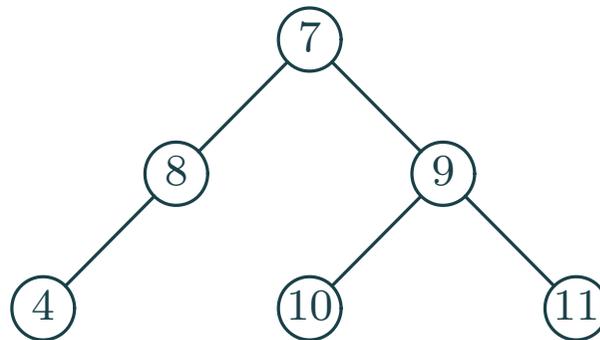
$A \rightarrow [ A \text{ nb } A ]$

$A \rightarrow$

arbre syntaxique  
(arbre de syntaxe  
concrète)



arbre de syntaxe  
abstraite



# Arbre syntaxique (2/5)

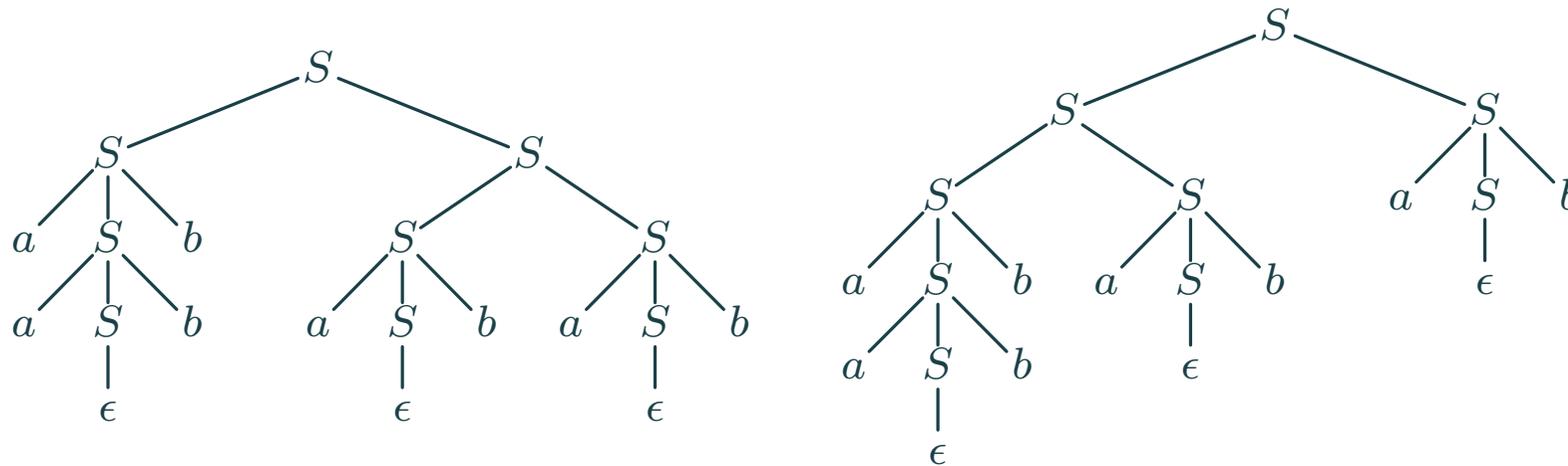
Mot *aabbabab*

Grammaire

$S \rightarrow SS$

$S \rightarrow aSb$

$S \rightarrow$



etc.

Langage parenthésé

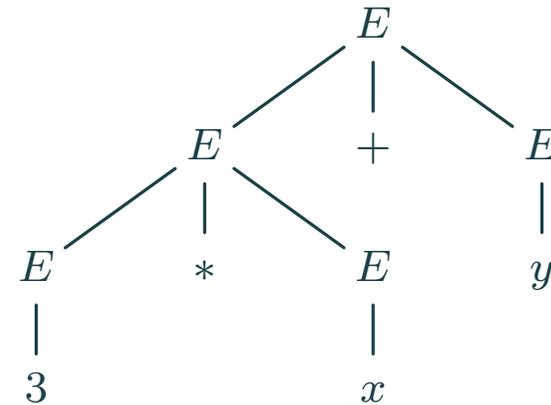
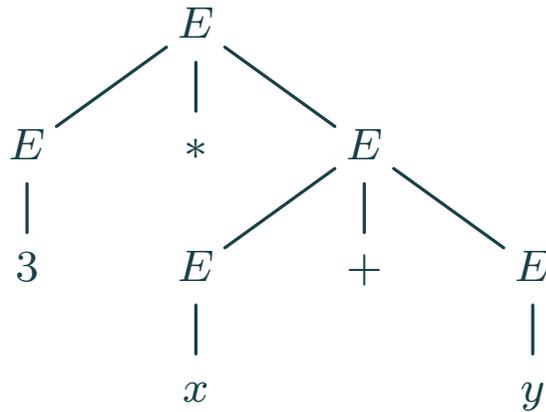
Une grammaire est **ambigüe** si elle génère plus d'un arbre syntaxique pour un même mot reconnu.

**Exercice 2** Trouver une grammaire non ambigüe générant le langage parenthésé précédent.

# Arbre syntaxique (3/5)

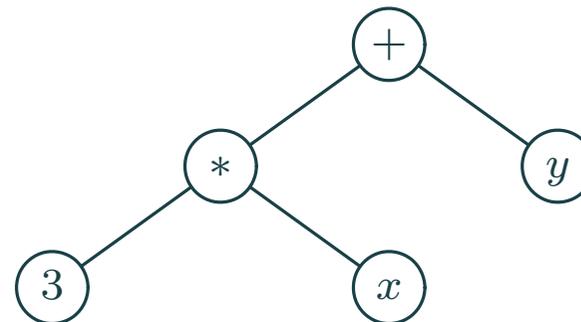
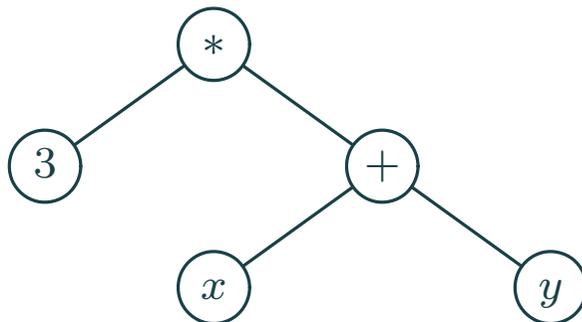
Mot  $3 * x + y$

Grammaire Expressions arithmétiques 1



Grammaire ambiguë, car deux arbres syntaxiques pour  $3 * x + y$

$\Rightarrow$  deux ASAs pour  $3 * x + y$

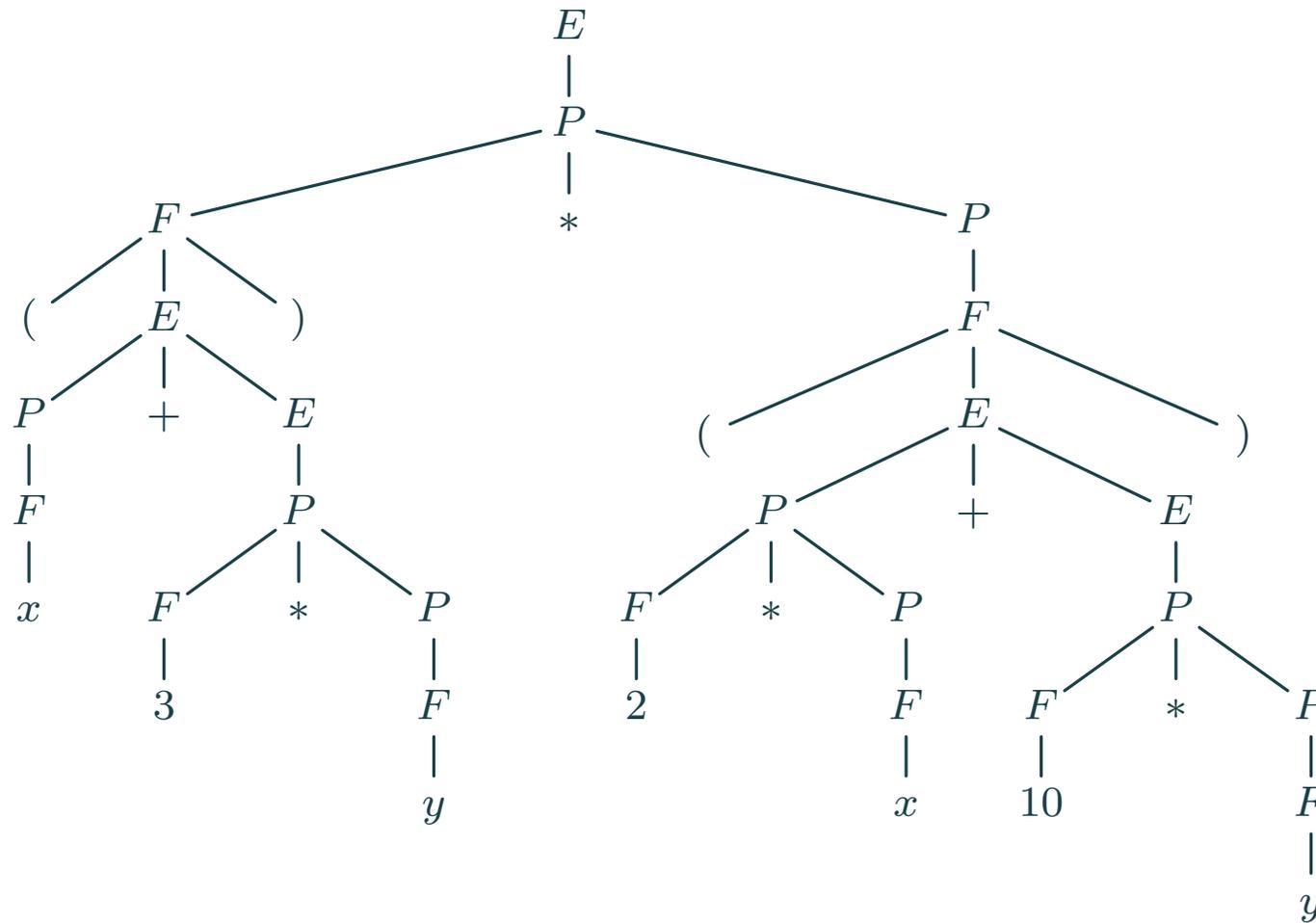


# Arbre syntaxique (4/5)

Mot  $(x + 3 * y) * (2 * x + 10 * y)$

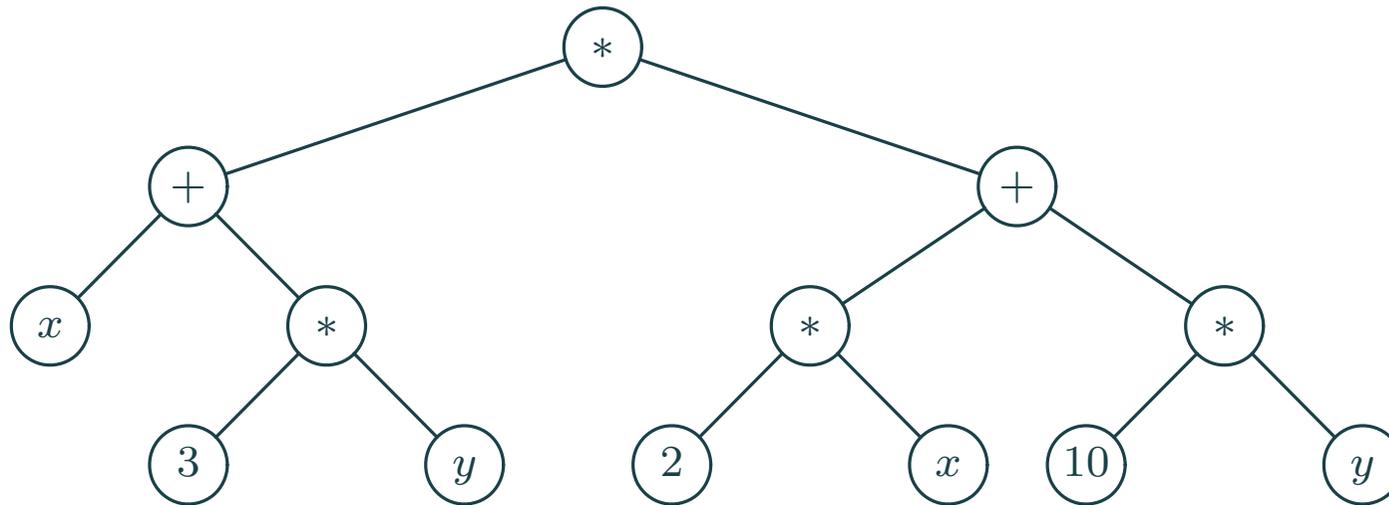
Expressions arithmétiques 2

$E \rightarrow P + E$      $E \rightarrow P - E$      $E \rightarrow P$      $P \rightarrow F * P$      $P \rightarrow F / P$   
 $P \rightarrow F$      $F \rightarrow id$      $F \rightarrow nb$      $F \rightarrow ( E )$



# Arbre syntaxique (5/5)

ASA associé à  $(x + 3 * y) * (2 * x + 10 * y)$



**Exercice 3** Dans les quatre grammaires précédentes, donner les grammaires non ambiguës. Démontrer le !

# Grammaires BNF (Backus-Naur Form) (1/2)

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 souvent des raccourcis pour simplifier la notation. Un bout de la BNF de Java :

ForStatement:

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

ForInit:

```
StatementExpressionList  
LocalVariableDeclaration
```

ForUpdate:

```
StatementExpressionList
```

StatementExpressionList:

```
StatementExpression  
StatementExpressionList , StatementExpression
```

- StatementExpression:
  - Assignment
  - PreIncrementExpression
  - PreDecrementExpression
  - PostIncrementExpression
  - PostDecrementExpression
  - MethodInvocation
  - ClassInstanceCreationExpression

AssignmentExpression:

- ConditionalExpression
- Assignment

Assignment:  
LeftHandSide AssignmentOperator AssignmentExpression

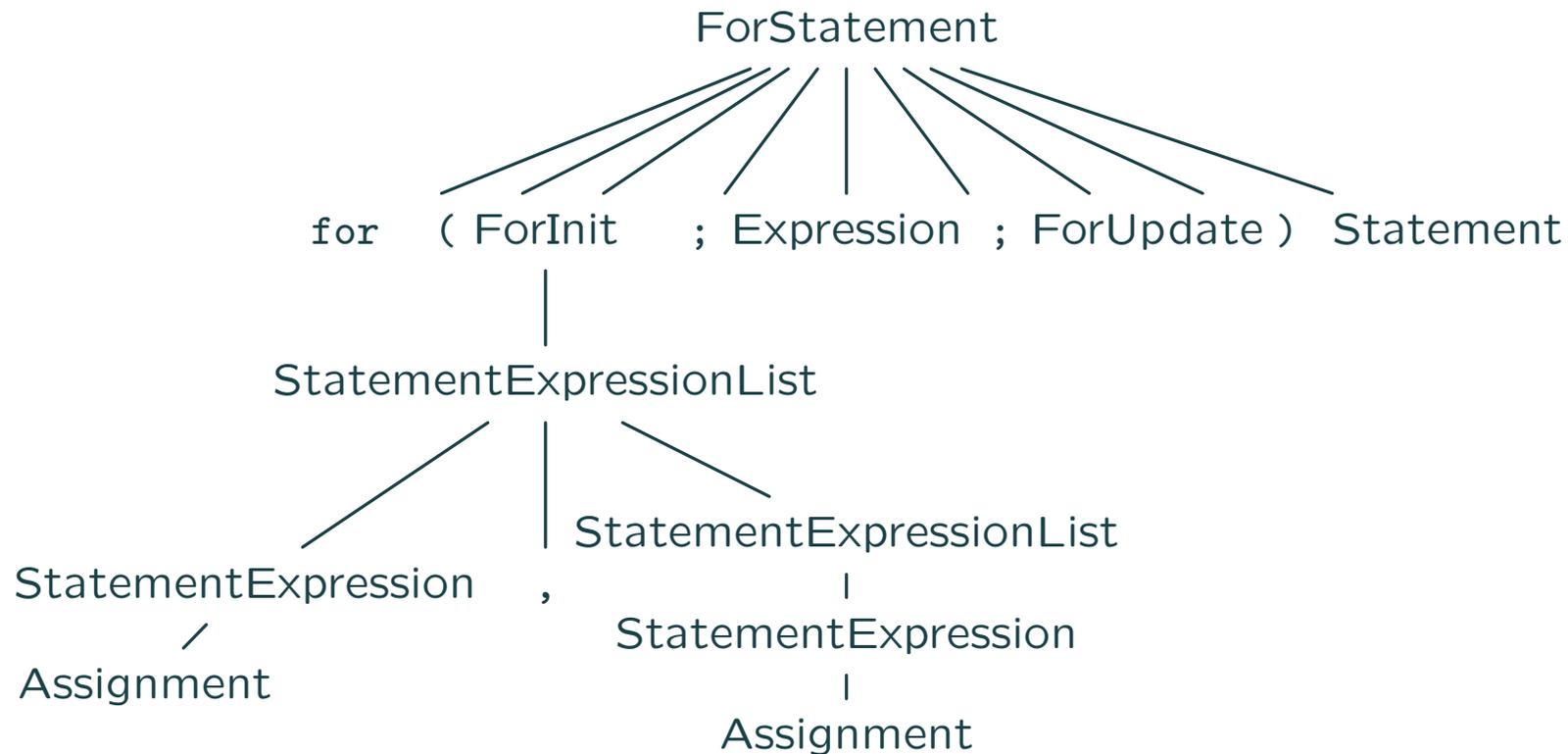
LeftHandSide:

- Name
- FieldAccess
- ArrayAccess

AssignmentOperator: **one of**  
= \*= /= %= += -= <<= >>= >>>= &= ^= |=

# Grammaires BNF (Backus-Naur Form) (2/2)

```
for (x = 1, y = 3; x < 100; ++x, ++y) {  
  Statement  
}
```



Parfois, on écrit les BNFs avec des diagrammes en chemin de fer.

# Analyse syntaxique

Donnés : grammaire  $G$  et un mot  $w$

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

2 grandes méthodes :

- analyse descendante. On part de  $S$  pour atteindre  $w$ .
- analyse ascendante. On part de  $w$  pour atteindre  $S$ .

déterministes (sans *backtracking*) pour des grammaires spéciales :

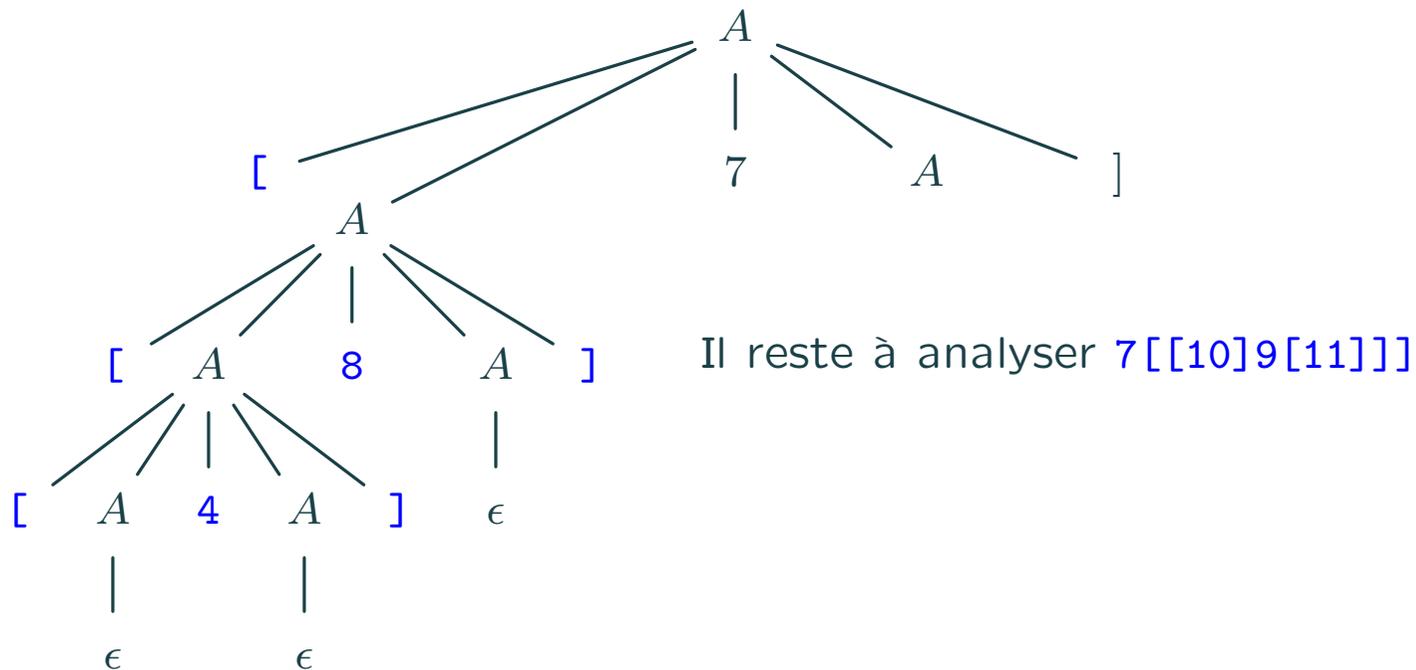
- grammaires  $LL(k)$  pour analyse descendante  
`javacc` ou Pascal [Wirth, 71]
- grammaires  $LR(k)$  pour analyse ascendante  
`yacc` [S. Johnson, 73]

Ici on verra l'analyse récursive descendante pour les grammaires  $LL(1)$ .

# Méthode récursive descendante (1/4)

Mot `[[[4]8]7[[10]9[11]]]`

On part de l'axiome  $A$  en choisissant l'une des 2 règles de production en fonction du premier lexème (terminal) de la chaîne d'entrée.



A la fin, on obtient l'arbre syntaxique de la planche 15. Animation 1

# Méthode récursive descendante (2/4)

Le lexème courant dans la variable globale lc.

```
static Lexeme lc;
static void avancer() {lc = Lexeme.suivant(); }

static Arbre lireArbre () {
    if (lc.nature == Lexeme.L_CroG) {
        avancer(); Arbre a = lireArbre();
        if (lc.nature == Lexeme.L_Nombre) {
            int x = lc.val;
            avancer(); Arbre b = lireArbre();
            if (lc.nature == Lexeme.L_CroD) {
                avancer(); return new Arbre (x, a, b);
            } else
                throw new Error ("Erreur de syntaxe. Il manque \"]\".");
        } else
            throw new Error ("Erreur de syntaxe. Il manque un nombre.");
    } else
        return null;
}
}
```

# Méthode récursive descendante (3/4)

Le lexème courant dans la variable globale `lc`.

```
static Lexeme lc;
static void avancer() {lc = Lexeme.suivant(); }

static Terme expression() {
    Terme t = produit(); switch (lc.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;
    }
}
```

- Une fonction (récursive) par variable non terminale.
- Au début, on appelle la fonction correspondant à l'axiome.

# Méthode récursive descendante (4/4)

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

```
static Terme facteur() {
    Terme t; switch (lc.nature) {
        case Lexeme.L_ParG: avancer(); t = expression();
            if (lc.nature != Lexeme.L_ParD) throw new Error ("Il manque ')'");
            break;
        case Lexeme.L_Nombre: t = new Terme (lc.val); break;
        case Lexeme.L_Id: t = new Terme (lc.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 les ASAs

- **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** (vérifier l'absence de débordements flottants dans le logiciel embarqué d'Ariane 5 **avant** son exécution)

# Evaluation d'expressions arithmétiques

Données :  $t$  terme,  $e$  table des valeurs des variables.

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

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

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

# Dérivation

Données :  $t$  terme,  $x$  nom de variable.

But : calcul la dérivée du terme  $t$  par rapport à  $x$ .

```
static int deriver(Terme t, String x) {
    switch (t.nature) {
    case ADD: return new Terme(ADD, deriver(t.a1, x), deriver(t.a2, x));
    case SUB: return new Terme(SUB, deriver(t.a1, x), deriver(t.a2, x));
    case MUL: return new Terme(ADD,
        new Terme(MUL, deriver(t.a1, x), t.a2),
        new Terme(MUL, t.a1, deriver(t.a2, x)));
    case DIV: return new Terme(DIV,
        new Terme(SUB,
            new Terme(MUL, deriver(t.a1, x), t.a2),
            new Terme(MUL, t.a1, deriver(t.a2, x))),
        new Terme(MUL, t.a2, t.a2));
    case CONST: return new Terme(CONST, 0);
    case VAR: if (t.nom.equals(x)) return new Terme(CONST, 1)
        else return new Terme(CONST, 0);
    }
}
```

Remarque : le résultat peut être un *dag* (dû au partage de sous-termes)

# Belle impression

```
static void impExp (Terme t) {  
    switch (t.nature) {  
        case ADD: impProd(t.a1); System.out.print("+"); impExp(t.a2); break;  
        case SUB: impProd(t.a1); System.out.print("-"); impExp(t.a2); break;  
        default: impProd(t);  
    }  
}
```

```
static void impProd (Terme t) {  
    switch (t.nature) {  
        case MUL: impFact(t.a1); System.out.print("*"); impProd(t.a2); break;  
        case DIV: impFact(t.a1); System.out.print("/"); impProd(t.a2); break;  
        default: impFact(t);  
    }  
}
```

```
static void impFact (Terme t) {  
    switch (t.nature) {  
        case CONST: System.out.print(t.val); break;  
        case VAR: System.out.print(t.nom); break;  
        default: System.out.print("("); impExp(t); System.out.print(")"); break;  
    }  
}
```

Belle symétrie par rapport à l'analyse syntaxique (opération inverse).

# Opérateurs non associatifs

La méthode récursive descendante parenthèse mal les opérateurs non associatifs.

$$x + y + z \quad \text{analysé comme} \quad x + (y + z)$$

$$x - y - z \quad \quad \quad x - (y - z)$$

Pour revenir au parenthésage naturel, implicitement à gauche, il faut transformer la grammaire en :

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

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

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

**Impossible** à analyser en récursif descendant (**récursivité gauche** dans la grammaire), puisque  $E \rightarrow^* Eu$ .

Avec des expressions régulières, on écrit la grammaire comme

$$E \rightarrow P ( + P )^* \quad E \rightarrow P ( - P )^*$$

$$P \rightarrow F ( * F )^* \quad P \rightarrow P ( / F )^*$$

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

et on utilise un while dans la programmation.

# Analyse syntaxique

- théorie des langages formels [Chomsky, Schutzenberger, 1960]
- analyse syntaxique [Aho, Sethi, Ullman, 1980]
- compilation [Appel], [Cam1, Ocaml, ...Leroy]
- analyse de langues naturelles

Les grammaires formelles  $\Rightarrow$  Automates et Calculabilité en majeure M1 ;

les analyseurs syntaxiques  $\Rightarrow$  Compilation en majeure M2.

# Exercices

**Exercice 4** Programmer la belle impression.

**Exercice 5** Essayer d'imaginer ce que pourrait être l'analyse ascendante.

**Exercice 6** Trouver une solution grammaticale LL(1) pour l'analyse syntaxique des opérateurs non associatifs.

**Exercice 7** Donner des exemples où l'analyse descendante a des difficultés, mais pas une analyse ascendante.

**Exercice 8** Faire une calculette logique

**Exercice 9** Faire une calculette Texas Instrument (sans notation polonaise).