INF 431

F. Morain

Analyse syntaxique

28 février 2007

Où en est-on ?

Amphi 1: introduction.

Amphi 2: génie logiciel avec Java.

Amphi 3: analyse lexicale.

Amphi 4: analyse syntaxique.

Amphi 5: graphes I.

Amphi 6: graphes II (parcours).

Amphi 7: graphes III (optimisation combinatoire).

Amphi 8: graphes IV (topologie).

Plan

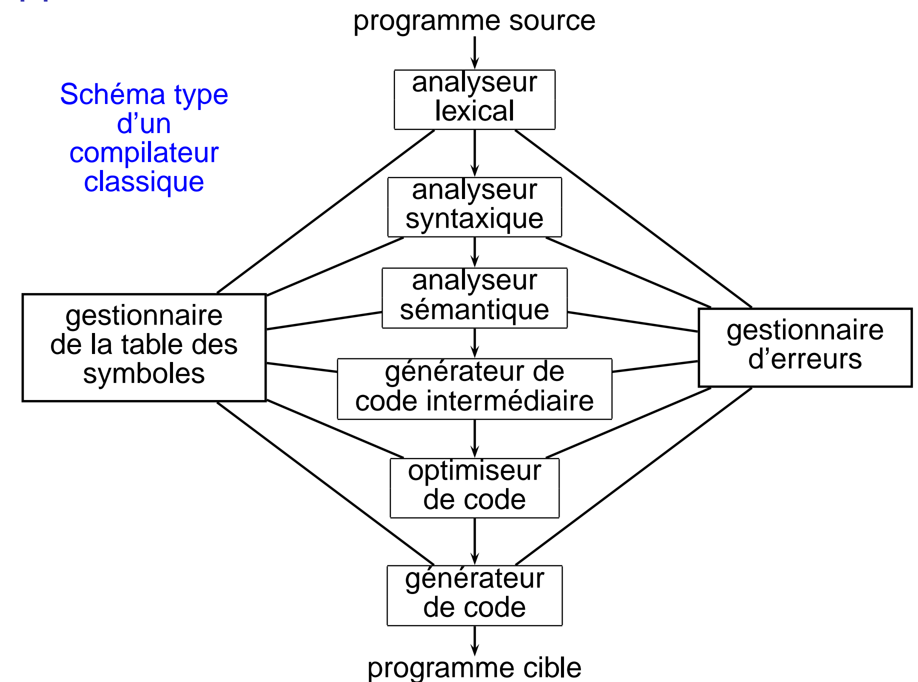
I. Arbres de syntaxe abstraite.

II. Génération de code.

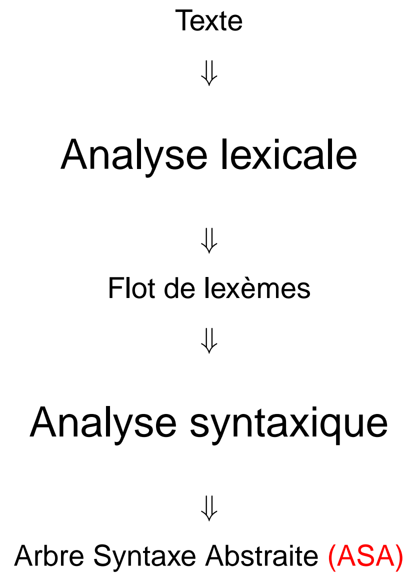
III. Grammaires formelles.

IV. Analyse syntaxique.

Rappel du contexte



Zoom

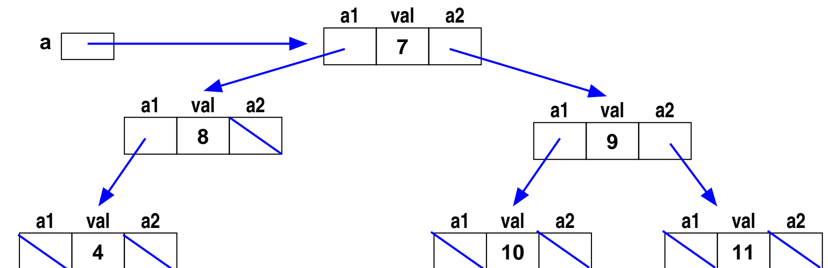


I. Arbres de syntaxe abstraite

- 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)
  - analyses statiques (vérification des programmes avant exécution – Ariane 5)
- Exemples d'ASAs:
  - arbres binaires
  - termes représentant une expression arithmétique

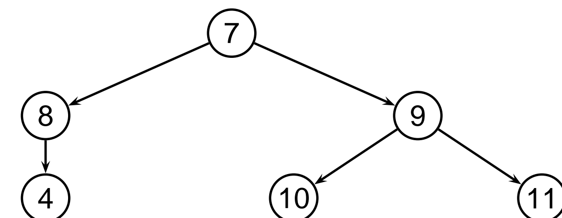
Rappels sur les arbres binaires

```
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;
  }
}
```

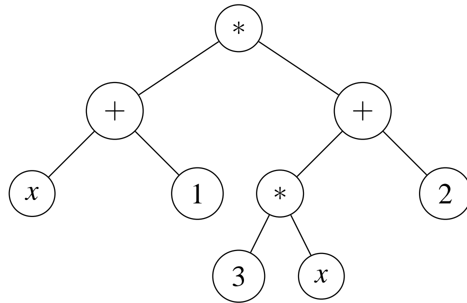

```
Arbre a = new Arbre(7,
            new Arbre(8, new Arbre(4), null),
            new Arbre(9,
              new Arbre(10), new Arbre(11)));
```

ou encore plus abstraitement



Termes

But: calculer. Les termes sont représentés par des arbres.
Ex. pour $(x+1)*(3*x+2)$



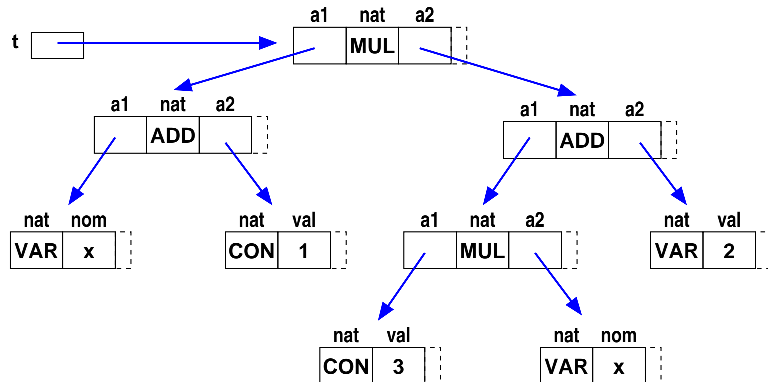
Exercice. Dessiner les ASA pour les termes $x+y+z$, $x*y*z$, $x*y+z$, $x*(y+z)$, $(a+a*a)*(a*a+a*a)$.

Implantation

```
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; }
}
```


Exemple (suite)

```
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.

Évaluation d'expressions arithmétiques

Données: t terme, e table des valeurs des variables (une liste de couples (nom, valeur)).
But: calculer 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 CONST: return new Terme(CONST, 0);
  case VAR: if(t.nom.equals (x))
              return new Terme(CONST, 1);
            else return new Terme(CONST, 0);
  case ADD:
       return new Terme(ADD,
                    deriver(t.a1, x),
                    deriver(t.a2, x));
  case SUB: ...
  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: ...
  }
}
```


Belle impression (sans parenthèse superflue)

```
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;
  }
}
```


II. Génération de code

Processeur simplifié: la machine a des registres R_i dans lesquels s'effectuent les opérations arithmétiques.
$R_i \leftarrow n$: chargement (*load*) d'une constante entière dans le registre R_i.
$R_i \leftarrow x$: chargement d'une constante entière depuis la mémoire x.
Autres opérations:

$$\begin{array}{lll} R_i \leftarrow n & R_k \leftarrow R_i + R_j & R_k \leftarrow R_i \times R_j \\ R_i \leftarrow x & R_k \leftarrow R_i - R_j & R_k \leftarrow R_i \,/\, R_j \end{array}$$

Ex.

```
a <- 2;
b <- a+1;
```

peut se compiler en

$$R_0 \leftarrow 2$$
$$R_1 \leftarrow 1$$
$$R_2 \leftarrow R_0 + R_1$$

Utilisation de l'ASA

Idée: on utilise l'ASA pour générer le code. On part d'un numéro de registre qu'on peut utiliser. On retourne le dernier utilisé:

```
compiler(A, i)
si A est une feuille f alors
  écrire "R[i] <- f"; // constante ou variable
  retourner i;
sinon
  A = (op, Ag, Ad);
  g <- compiler(Ag, i);
  d <- compiler(Ad, g+1);
  écrire "R[d+1] <- R[g] op R[d]";
  retourner d+1;
```


Sur l'ASA déjà vu:

$R_0 \leftarrow x$	$R_0 \leftarrow x$
$R_1 \leftarrow 1$	$R_1 \leftarrow 1$
$R_2 \leftarrow R_0 + R_1$	$R_0 \leftarrow R_0 + R_1$
$R_3 \leftarrow 3$	$R_1 \leftarrow 3$
$R_4 \leftarrow x$	$R_2 \leftarrow x$
$R_5 \leftarrow R_3 * R_4$	$R_1 \leftarrow R_1 \times R_2$
$R_6 \leftarrow 2$	$R_2 \leftarrow 2$
$R_7 \leftarrow R_5 + R_6$	$R_1 \leftarrow R_1 + R_2$
$R_8 \leftarrow R_2 * R_7$	$R_0 \leftarrow R_0 \times R_1$

Rem. Très couteux en nombre de registres. Un processeur n'en a jamais beaucoup $\Rightarrow$ on doit faire mieux.

Nombre minimal de registres (à la Ershov)

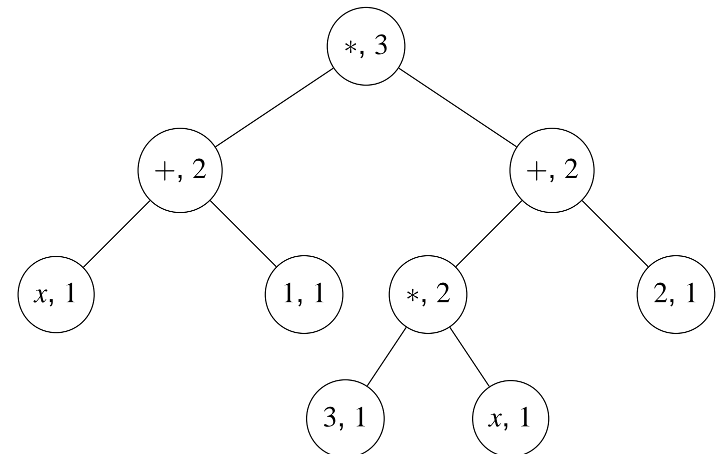
- évaluer une constante ou une variable: 1 registre;
- pour évaluer $e \oplus e'$:
  - si e et e' ont besoin d'un même nombre de registres, on a besoin d'un registre de plus pour le résultat;
  - on évalue e avec le plus grand nombre de registres, et on stocke le résultat dans un registre non utilisé par e'.

$$reg(x) = reg(n) = 1$$

$$reg(e \oplus e') = \begin{cases} 1 + reg(e) & \text{si } reg(e) = reg(e') \\ \max(reg(e), reg(e')) & \text{sinon} \end{cases}$$

Nombres de Horton-Strahler: $reg(e)$ ont un correspondant en hydrologie et en botanique. Ils donnent l'épaisseur d'une rivière en fonction de l'épaisseur de ses affluents, ou d'une branche en fonction de sa structure de branchement.

Exemple



Le code

```
static int reg(Terme t){
  switch(t.nature){
  case ADD: case SUB: case MUL: case DIV:
    int n1 = reg(t.a1), n2 = reg(t.a2);
    if (n1 == n2) return 1 + n1;
    else return Math.max(n1,n2);
  case CONST: case VAR: return 1;
  default:
    throw new Error("Erreur dans evaluation");
  }
}
```

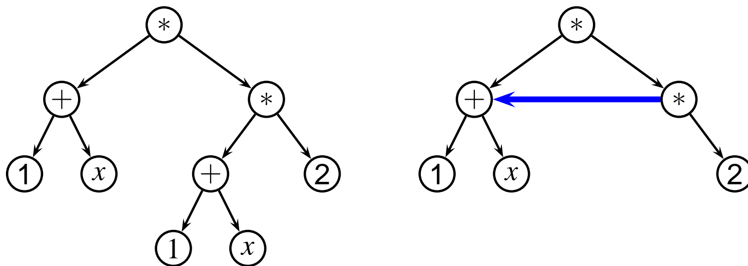


```
static int genCode(Terme t, int r0){
  switch (t.nature){
  case ADD: case SUB: case MUL: case DIV:
    int n1 = reg(t.a1), n2 = reg(t.a2);
    if(n1 >= n2){
      int r1=genCode(t.a1,r0), r2=genCode(t.a2,r1);
      System.out.println("R"+r1+"<-R"+r1+"+R"+r2);
      return r1;
    } else {
      int r2=genCode(t.a2,r0), r1=genCode(t.a1,r2);
      System.out.println("R"+r2+"<-R"+r2+"+R"+r1);
      return r2;
    }
  case CONST:
    System.out.println("R"+r0+"<-"+t.val);
    return r0;
  case VAR:
    System.out.println("R"+r0+"<-"+t.nom);
    return r0;
  default: throw new Error("Erreur dans evaluation");
  }
}
```


Remarques complémentaires

Cf. cours de compilation (Majeure2) pour des stratégies moins naïves:

- utiliser le partage (les dags):



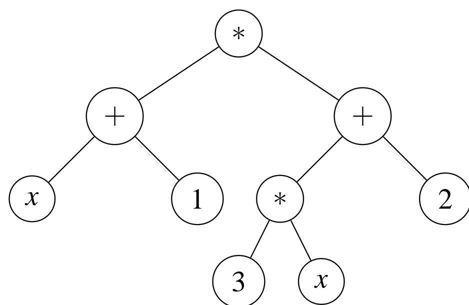
- utiliser des registres spéciaux;
- durée de vie des variables.

III. Grammaires formelles

Problématique: on dispose de mots écrits sur un alphabet, qu'on assemble à l'aide de règles (une grammaire) pour obtenir une phrase (expression).

On souhaite élaborer une méthode systématique qui transforme une phrase en ASA, de façon automatique.

Ex. pour $(x+1)*(3*x+2)$, on a un alphabet composé de lettres et chiffres, des opérateurs, et des règles de combinaison entre opérateurs. On veut en tirer :



L'écriture $(x+1)*(3*x+2)$ est habituelle, mais on pourrait aussi l'écrire (formes polonaises) :

- en notation préfixée : $*+x1+*3x2$
- en notation postfixée : $x1+3x*2+*$

Ces deux formules sont très faciles à analyser pour construire l'ASA.

Construction dans le cas préfixé (esquisse)

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

static Terme lireExpr(){
  if(lc.nature == Lexeme.L_Mul){
    avancer(); Terme g = lireExpr();
    avancer(); Terme d = lireExpr();
    return new Terme(MUL, g, d);
  }
  else if(lc.nature == Lexeme.L_Plus){ ... }
  else if(lc.nature == Lexeme.L_Nombre){
    return new Terme(lc.valeur);
  }
  else if(lc.nature == Lexeme.L_Id){
    return new Terme(lc.nom);
  }
  else{ ... }
}
```

Ce cas est simple, mais peut-être n'est-ce pas ce dont rêve le client. . .

Grammaires algébriques

Grammaires algébriques ou hors-contexte (*context-free*) au sens de Chomsky.

$G=(\Sigma,V,S,\mathscr{P})$ où

- Σ alphabet terminal
- V ensemble fini des variables non terminales ($V\cap\Sigma=\emptyset$)
- S axiome ($S\in V$)
- $\mathscr{P}$ ensemble fini de règles de productions $X_i\rightarrow w_i$ avec $X_i\in V$ et $w_i\in(V\cup\Sigma)^*$.

Ex. (Mots correctement parenthésés) $\Sigma=\{a,b,x\}$, $V=\{S\}$, avec les règles:

$$S\rightarrow x \quad S\rightarrow aSbS \quad S\rightarrow$$

(penser que $a=$ **(** et $b=$ **)**).

Déf. Soient w, w' dans $(V\cup\Sigma)^*$. On dit que w se récrit en w' (ou que w' dérive en un coup de w), ce que l'on note $w\rightarrow_G w'$ (ou $w\rightarrow w'$) ssi :

1. w se décompose en $w=svt$ avec $s,t\in(V\cup\Sigma)^*$ et $v\in V$;
2. w' se décompose en $w'=sft$ avec les même s,t et $f\in(V\cup\Sigma)^*$;
3. (v,f) est une règle de G.

Ex. $w=aSb\rightarrow w'=axb$; $aSb\rightarrow aaSbSb$.

Langage reconnu par G:

- $v\rightarrow v'$ implique $uvw\rightarrow uv'w$;
- dérivation: $u\rightarrow^* v$ ssi $u=u_0\rightarrow u_1\rightarrow u_2\rightarrow\cdots u_n=v$ ($n\geqslant 0$); terminale si $v\in\Sigma^*$;
- $L(G)=\{w\in\Sigma^*\mid S\rightarrow^* w\}$.

Exemple

Avec:

$$S \to aSb, \quad S \to \varepsilon$$

On peut obtenir typiquement:

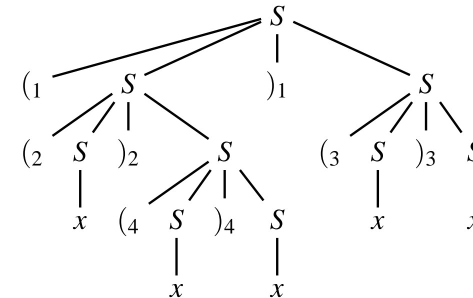
$$S \to aSb \to aaSbb \to aaaSbbb \to aaabbb,$$

et on a caractérisé $L = \{a^n b^n, n \geqslant 0\}$, dont on peut montrer qu'il n'est pas reconnaissable par un automate fi ni (cf. poly).

Arbre de dérivations

On associe à une suite de dérivations un arbre de dérivation.

$$S \to (S)S \to (S)(S)S \to (S)(x)S \to ((S)S)(x)S \to ((S)(S)S)(x)S$$
$$\to ((S)(x)S)(x)S \to ((S)(x)x)(x)S \to ((S)(x)x)(x)x \to ((x)(x)x)(x)x,$$



Les mots obtenus à chaque étape de la dérivation sont les lisières des arbres de dérivation successifs; les mots du langage sont les lisières écrites sur l'alphabet terminal.
Lisière $=$ le mot obtenu en lisant les feuilles de l'arbre en ordre préfi xe.

Un même mot peut posséder plusieurs dérivations :

$$S \to (S)S \to ((S)S)S \to ((x)S)S \to ((x)(S)S)S \to ((x)(x)S)S$$
$$\to ((x)(x)x)S \to ((x)(x)x)(S)S \to ((x)(x)x)(x)S \to ((x)(x)x)(x)x.$$

Ici, en récrivant le plus symbole le plus à gauche dans l'arbre.

Prop. Toute dérivation dans une grammaire hors-contexte est équivalente à une dérivation dans laquelle le symbole non-terminal le plus à gauche est systématiquement récrit.

$\Rightarrow$ Les récritures ne dépendent pas du contexte.

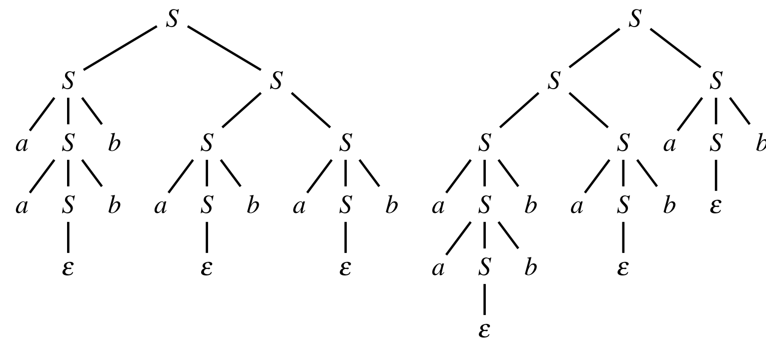
$\Rightarrow$ Chaque sous-arbre de l'arbre de dérivation peut être calculé indépendamment des autres.

Ambiguïté

Déf. Une grammaire est dite non-ambiguë ssi chaque mot du langage $L(G)$ possède un arbre de dérivations unique.

Ex 1. La grammaire écrite pour $\{a^n b^n \mid n \geqslant 0\}$ est non ambiguë.

Ex 2. La grammaire : $S \to SS \quad S \to aSb \quad S \to$ est ambiguë, puisque pour le mot $aabbabab$, on a



Autres exemples de grammaire

Représentation linéaire d'arbres

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

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

$A \rightarrow$

Expressions arithmétiques 1

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

$E \rightarrow E + E \quad E \rightarrow E - E \quad E \rightarrow E * E \quad E \rightarrow E\ /\ E$

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

Expressions arithmétiques 2

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

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

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

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

Dérivations avec les arbres

Représentation linéaire des arbres binaires

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

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

$A \rightarrow$

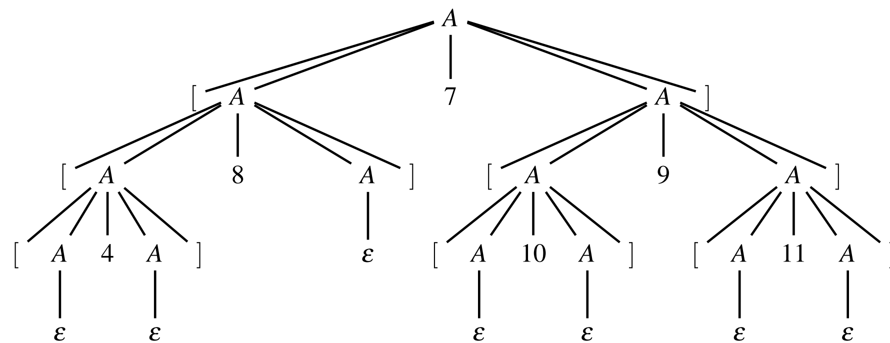
$A \rightarrow [\ A7A\ ] \rightarrow [\ [\ A8A\ ]\ 7A\ ] \rightarrow [\ [\ [\ A4A\ ]\ 8A\ ]\ 7A\ ]$
$\rightarrow [\ [\ [\ 4A\ ]\ 8A\ ]\ 7A\ ] \rightarrow [\ [\ [\ 4\ ]\ 8A\ ]\ 7A\ ] \rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7A\ ]$
$\rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ A9A\ ]\ ] \rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ A10A\ ]\ 9A\ ]\ ]$
$\rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ 10A\ ]\ 9A\ ]\ ] \rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ 10\ ]\ 9A\ ]\ ]$
$\rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ 10\ ]\ 9\ [\ A11A\ ]\ ]\ ]$
$\rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ 10\ ]\ 9\ [\ 11A\ ]\ ]\ ]$
$\rightarrow [\ [\ [\ 4\ ]\ 8\ ]\ 7\ [\ [\ 10\ ]\ 9\ [\ 11\ ]\ ]\ ]$

Long mais précis !
On peut abstraire les dérivations par un arbre de dérivations ou encore arbre syntaxique.

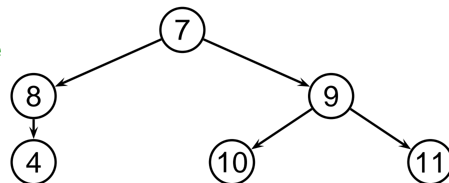
Arbre syntaxique

Mot: `[[[4]8]7[[10]9[11]]]` Grammaire: $A \rightarrow [\ A\ nb\ A\ ] \quad A \rightarrow$

arbre syntaxique (arbre de syntaxe concrète)

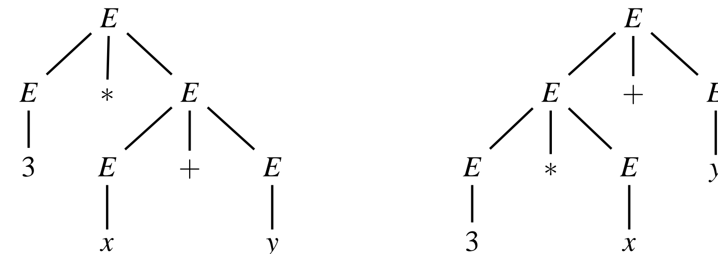


arbre de syntaxe abstraite

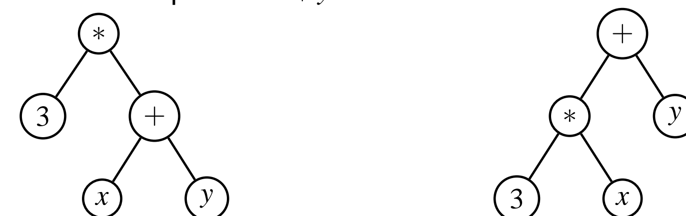


Expressions arithmétiques

Mot: $3 * x + y$ Grammaire: Expressions arithmétiques 1



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

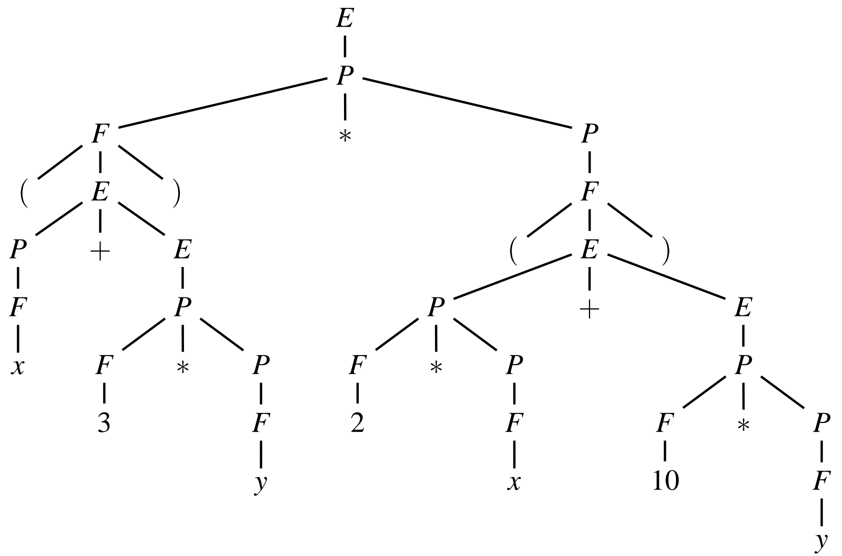


Expressions arithmétiques (suite)

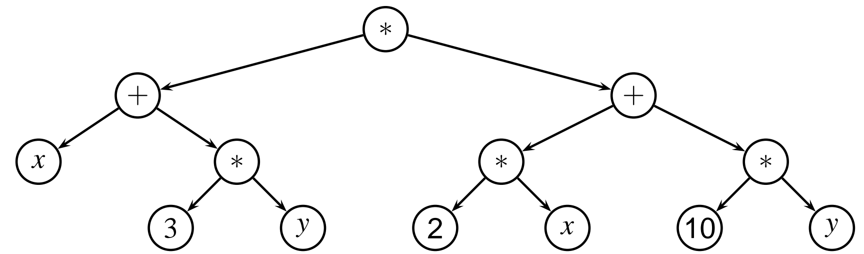
Mot $(x+3*y)*(2*x+10*y)$ Expressions arithmétiques 2

$E \to P + E$ $E \to P - E$ $E \to P$ $P \to F * P$ $P \to F \,/\, P$

$P \to F$ $F \to id$ $F \to nb$ $F \to (\ E\ )$



L'ASA associé à $(x+3*y)*(2*x+10*y)$



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

ForStatement:
    `for (` **ForInit**$_{opt}$ `;` **Expression**$_{opt}$ `;` **ForUpdate**$_{opt}$ `)`
        Statement

ForInit:
    StatementExpressionList
    LocalVariableDeclaration

ForUpdate:
    StatementExpressionList

StatementExpressionList:
    StatementExpression
    StatementExpressionList `,` **StatementExpression**

StatementExpression:
    Assignment
    PreIncrementExpression
    PreDecrementExpression
    PostIncrementExpression
    PostDecrementExpression
    MethodInvocation
    ClassInstanceCreationExpression

AssignmentExpression:
    ConditionalExpression
    Assignment

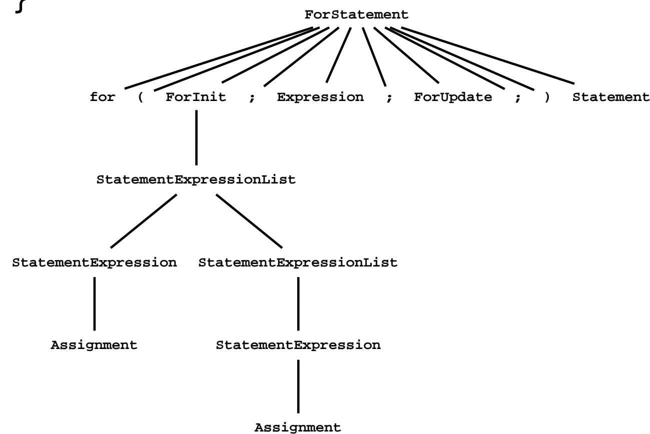
Assignment:
    LeftHandSide AssignmentOperator AssignmentExpression

LeftHandSide:
    Name
    FieldAccess
    ArrayAccess

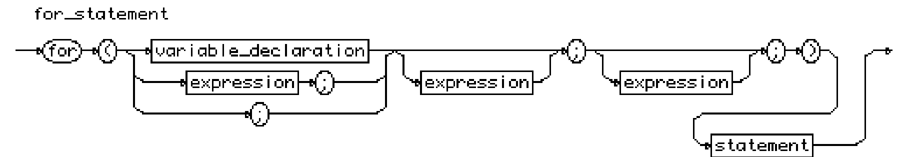
AssignmentOperator: **one of**
    `= *= /= %= += -= <<= >>= >>>= &= ^= |=`

Exemple avec Java

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



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



IV. 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]

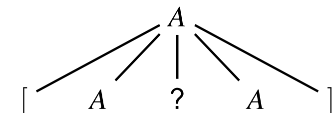
Ici on verra l'analyse récursive descendante pour les grammaires $LL(1)$.

Méthode récursive descendante

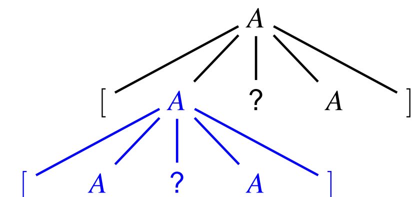
Mot `[[[4]8]7[[10]9[11]]]`

Principe: 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.

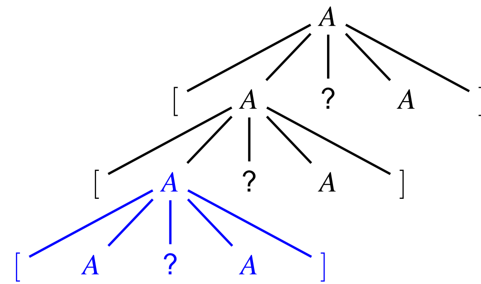
`[[[4]8]7[[10]9[11]]]`



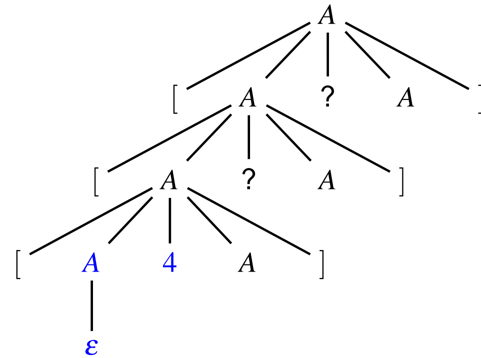
`[[[4]8]7[[10]9[11]]]`



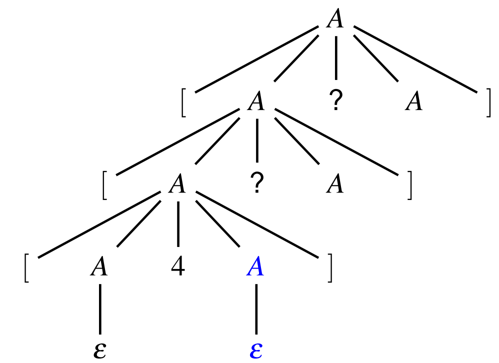
`[[[4]8]7[[10]9[11]]]`



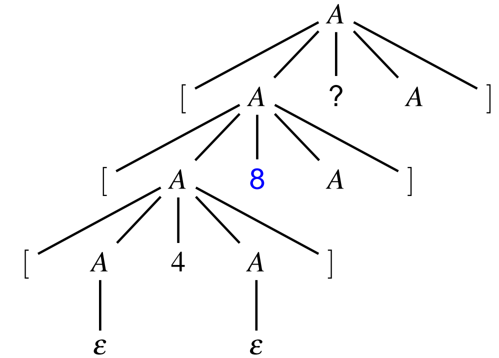
`[[[4]8]7[[10]9[11]]]`



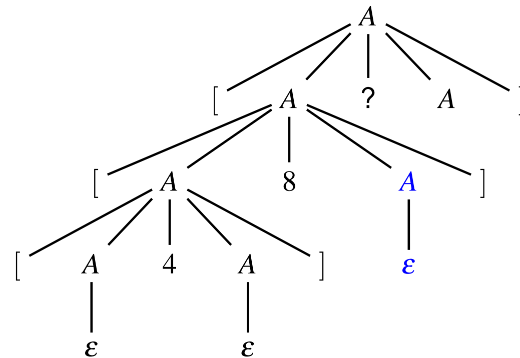
`[[[4]8]7[[10]9[11]]]`



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



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



Il reste à analyser `7[[10]9[11]]]`, etc.

À la fin, on obtient l'arbre syntaxique de la planche 35.

Implantation

```
static Lexeme lc; // lexème courant
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("Syntaxe: \"]\" manquant.");
    } else
      throw new Error("Syntaxe: nombre manquant.");
  }
  else return null;
}
```

Retour aux termes: expressions arithmétiques 1

Principes:

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

```
static Lexeme lc; // lexème courant
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;
  }
}
```


On décide toujours avec au plus un caractère d'avance $LL(1)$.

```
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;
}
```


Opérateurs non associatifs

La méthode récursive descendante parenthèse mal les opérateurs non associatifs.

$x+y+z$	analysé comme	$x+(y+z)$
$x-y-z$		$x-(y-z)$

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

$$E \to E + P \qquad E \to E - P \qquad E \to P$$
$$P \to P * F \qquad P \to P \,/\, F \qquad P \to F$$
$$F \to id \qquad F \to nb \qquad F \to (\ E\ )$$

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

Avec des expressions régulières, on écrit la grammaire comme

$$E \to P\ (\ +\ P)^* \qquad E \to P\ (\ -\ P)^*$$
$$P \to F\ (\ *\ F)^* \qquad P \to P\ (\ /\ F)^*$$
$$F \to id \qquad F \to nb \qquad F \to (\ E\ )$$

Résumé du cours

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

La Compilation est traitée en majeure M2.

Prochains rendez-vous

- TD cet après-midi;
- mercredi prochain (07/03): graphes I.