Théorie des Langages

Analyses lexicale et syntaxique

Claude Moulin

Université de Technologie de Compiègne

Printemps 2013

Sommaire

Sommaire

Introduction

a i r e : = b a s e * h a u t e u r / 2

aire	:=	base	*	hauteur	/	2

lettre $\longrightarrow$ [A-Za-z]
chiffre $\longrightarrow$ [0-9]
chiffres $\longrightarrow$ chiffre (chiffre)*
fraction $\longrightarrow$. chiffres
exposant $\longrightarrow$ E(+ | - | ϵ) chiffres

Introduction

a i r e : = b a s e * h a u t e u r / 2

aire	:=	base	*	hauteur	/	2

identificateur $\longrightarrow$ lettre (lettre | chiffre)*
nombre $\longrightarrow$ chiffres fraction ? exposant ?
op-affectation $\longrightarrow$:=
op-relation $\longrightarrow$ = | < | > | <> | <= | >=
op-arithmétique $\longrightarrow$ * | / | + | -

Introduction

a i r e : = b a s e * h a u t e u r / 2

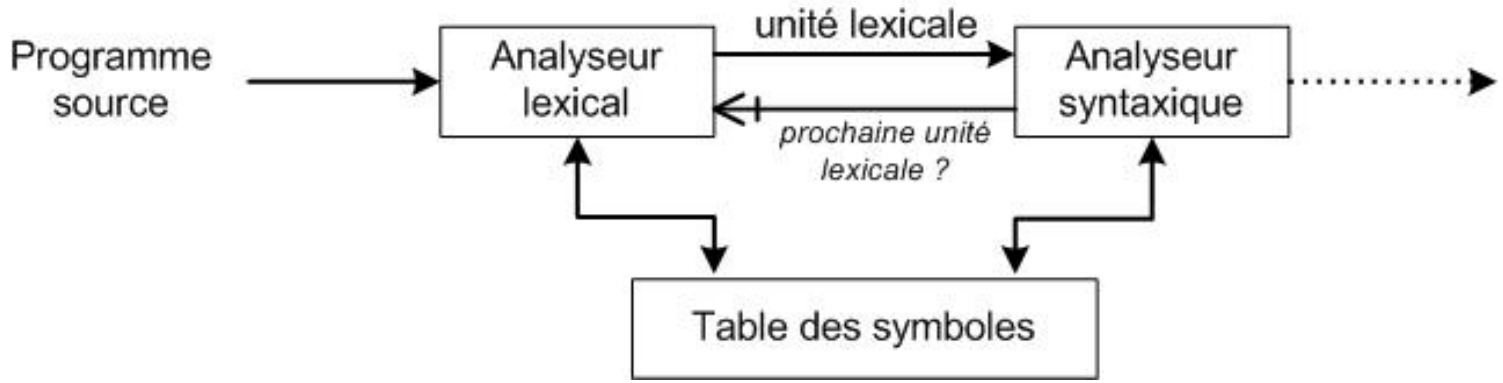
aire	:=	base	*	hauteur	/	2

<identificateur>
<op-affectation>
<identificateur>
<op-arithmétique>
<identificateur>
<op-arithmétique>
<nombre>

Unités lexicales

Lexèmes	Unité lexicale	Attribut
aire	identificateur	pointeur vers la table des symboles
:=	op-affectation	aucun
base	identificateur	pointeur vers la table des symboles
*	op-arithmétique	code
<	op-relation	code

Rôle de l'analyseur lexical



- lit les caractères d'entrée et produit une suite d'unités lexicales
- est lié à l'analyseur syntaxique
- initialise la table des symboles.
- élimine les commentaires et les séparateurs.
- relie les erreurs de compilation au programme source.

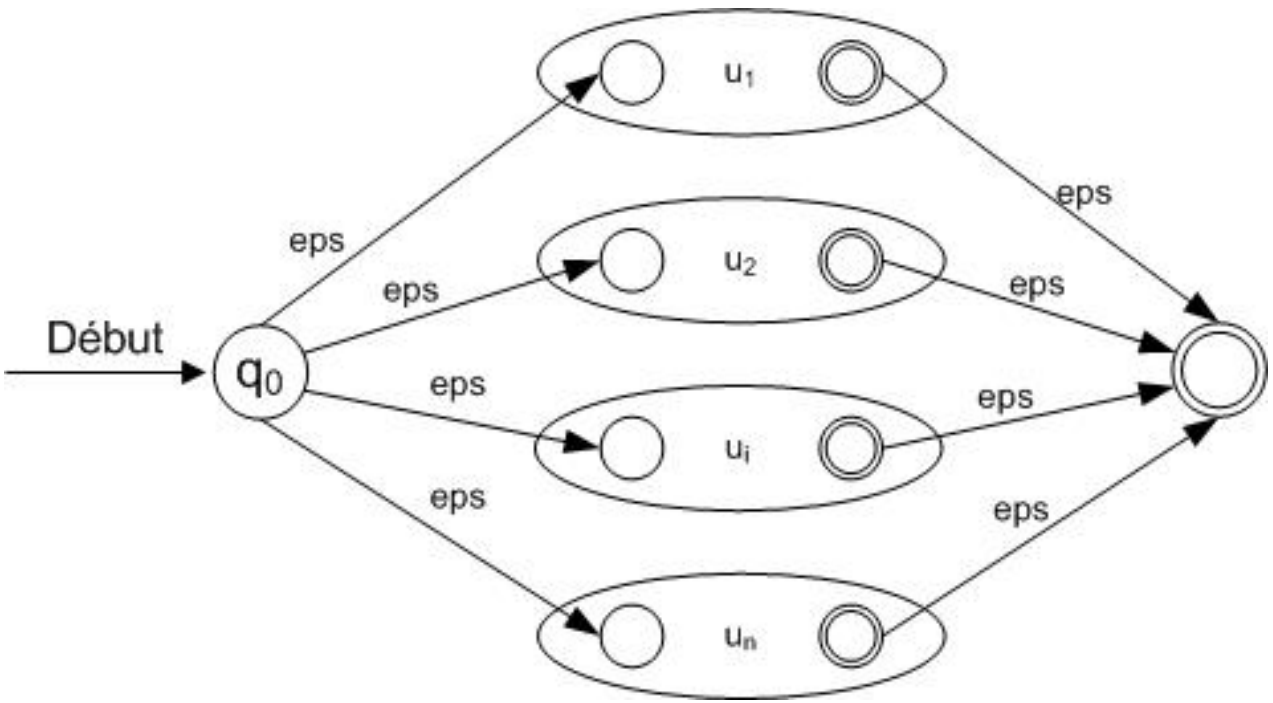
Sommaire

Définitions

- Un lexème est une chaîne de caractères.
- Une unité lexicale est un type de lexèmes (pour la syntaxe).
- Un modèle est une règle décrivant les chaînes qui correspondent à une unité lexicale.
- Un attribut est une information additionnelle (pour la sémantique)

Construction

- Les unités lexicales sont définies par des expressions régulières.
- A chaque unité lexicale u_1, u_2, ...u_n est associé un AFN.
- Un AFN général est bâti à partir de ces AFN de base par réunion.



Ambiguïté

- Plusieurs modèles peuvent représenter le même lexème
  - Caractères de prévision
    L'analyseur lexical lit des caractères en avant afin de déterminer l'unité lexicale correspondante.
  - Le prochain lexème le plus long est celui qui vérifie une expression régulière.
  - Le premier modèle (la première expression régulière) qui est vérifié détermine l'unité lexicale du léxème (Il y a un ordre dans l'écriture des définitions régulières).

Mots-clés

- Les mots clés sont des mots réservés et ne peuvent servir d'identificateurs :
  - On ne définit pas un modèle particulier.
  - On les traite a priori comme des identificateurs.
  - On recherche, en état d'acceptation, à l'aide de l'exécution de code, si un "identificateur" fait partie d'une table prédéfinie.
- Exemples :
  - if8 est un identificateur (règle du plus long).
  - if est un mot réservé.
  - if 8 donne deux lexèmes et produit une erreur syntaxique (une parenthèse est en général requise après if).

Mots-clés

- Les mots clés sont des mots réservés et ne peuvent servir d'identificateurs :
  - On ne définit pas un modèle particulier.
  - On les traite a priori comme des identificateurs.
  - On recherche, en état d'acceptation, à l'aide de l'exécution de code, si un "identificateur" fait partie d'une table prédéfinie.
- Exemples :
  - if8 est un identificateur (règle du plus long).
  - if est un mot réservé.
  - if 8 donne deux lexèmes et produit une erreur syntaxique (une parenthèse est en général requise après if).

Sommaire

Sommaire

Passage Lexer Parser

- On définit le vocabulaire : alphabet du lexer.
- On définit des expressions régulières pour représenter les catégories lexicales : variables du lexer.
- On définit une grammaire pour l'analyse syntaxique.
- Les variables du lexer servent de terminaux à la grammaire.

Exemple

Alphabet du parser

<identificateur>
<op-affectation>
<op-arithmétique>
<op-relation>
<nombre>

Règle du parser :
Expression $\longrightarrow$ <nombre>
Expression $\longrightarrow$ Expression <op-arithmétique> <nombre>
Affectation $\longrightarrow$ <identificateur> <op-affectation> Expression

Exemple - 1

- Un langage est défini par deux types d'instruction :
  - av 100 : av suivi par un nombre entier
  - td 90 : td suivi par un entier
- L'alphabet du lexer : a v t d 0 1 2 3 4 5 6 7 8 9 espace.
- Les variables du lexer : AV TD INT
- Les règles du lexer :
  - AV $\longrightarrow$ av
  - TD $\longrightarrow$ td
  - INT $\longrightarrow$ [0-9]+
- Les espaces servent de séparateur et n'on pas d'intérêt pour l'analyseur syntaxique.

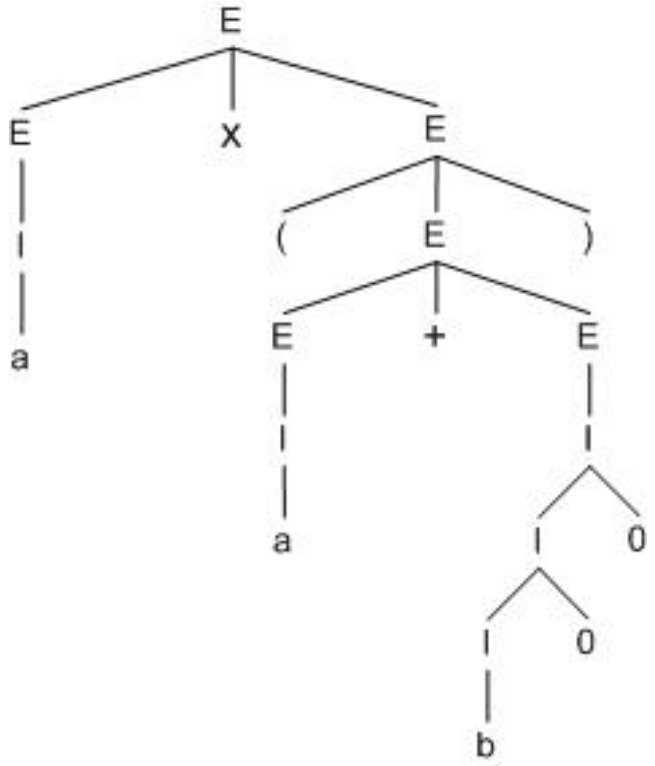
Exemple - 2

- programme $\longrightarrow$ liste_instructions
- liste_instructions $\longrightarrow$ instruction liste_instructions
- liste_instructions $\longrightarrow \epsilon$
- instruction $\longrightarrow$ AV INT
- instruction $\longrightarrow$ TD INT

Type d'analyse

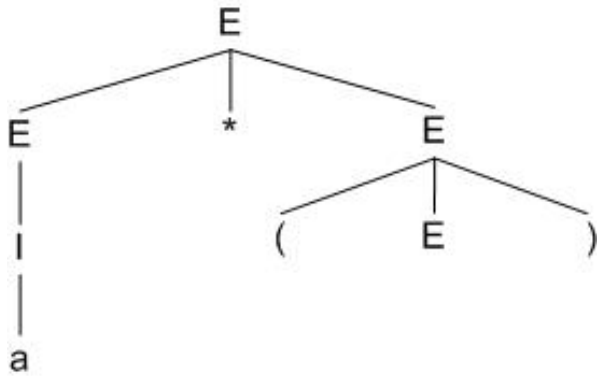
- L'analyse d'un programme se fait toujours en lisant les caractères un à un à partir du début de celui-ci.
- Le but est d'établir l'arbre de dérivation :

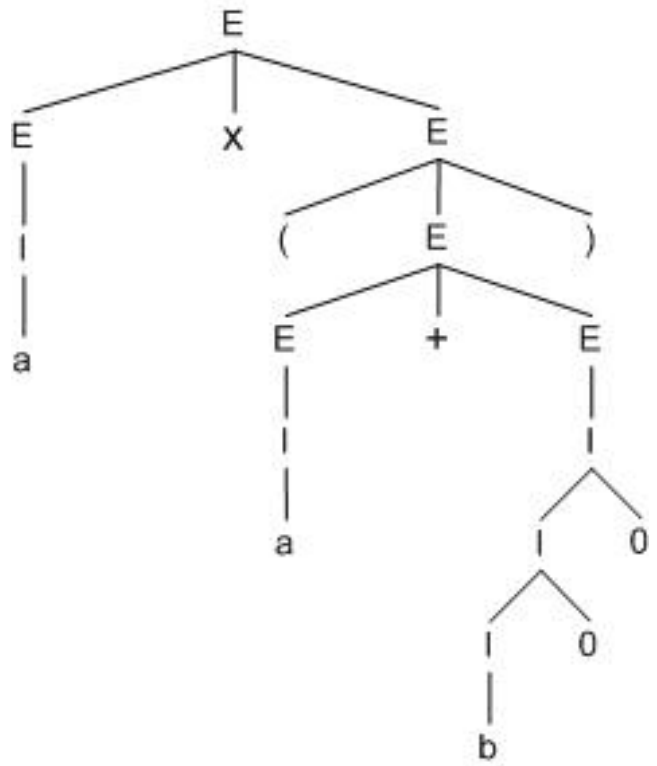
$a * (a + b00)$



Méthode LL

- Construction à partir de la racine

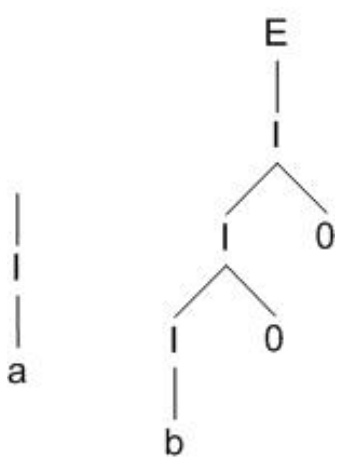


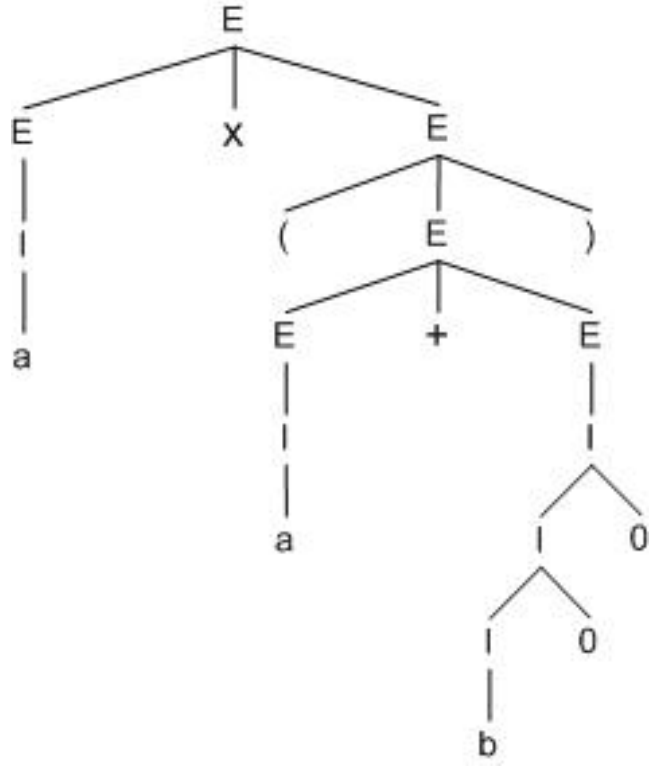


$$a * (a + b00)$$

Méthode LR

- Construction à partir des feuilles





$$a * (a + b00)$$

Exemple

$T = \{a, c, d\}$ $V = \{S, T\}$
$P = \{$

- 1. $S \longrightarrow aSbT$ 2. $S \longrightarrow cT$
- 3. $S \longrightarrow d$ 4. $T \longrightarrow aT$
- 5. $T \longrightarrow bS$ 6. $T \longrightarrow c$

$\}$
Chaîne : $w = accbbadbc$

Exemple

$T = \{a, c, d\}$ $V = \{S, T\}$
$P = \{$

- 1. $S \longrightarrow aSbT$ 2. $S \longrightarrow cT$
- 3. $S \longrightarrow d$ 4. $T \longrightarrow aT$
- 5. $T \longrightarrow bS$ 6. $T \longrightarrow c$

$\}$
Chaîne : $w = accbbadbc$
Analyse LL :
$S \underset{1}{\Longrightarrow} aSbT \underset{2}{\Longrightarrow} acTbT \underset{6}{\Longrightarrow} accbT \underset{5}{\Longrightarrow} accbbS \underset{1}{\Longrightarrow} accbbaSbT \underset{3}{\Longrightarrow} accbbadbT \underset{6}{\Longrightarrow} accbbadbc$

Exemple

$T = \{a, c, d\}$ $V = \{S, T\}$
$P = \{$

- 1. $S \longrightarrow aSbT$ 2. $S \longrightarrow cT$
- 3. $S \longrightarrow d$ 4. $T \longrightarrow aT$
- 5. $T \longrightarrow bS$ 6. $T \longrightarrow c$

$\}$
Chaîne : $w = accbbadbc$
Analyse LR :

$$accbbadbc \underset{6}{\Leftarrow} acTbbadbc \underset{2}{\Leftarrow} aSbbadbc \underset{3}{\Leftarrow} aSbbaSbc \underset{6}{\Leftarrow} aSbbaSbT \underset{1}{\Leftarrow} aSbbS \underset{5}{\Leftarrow} aSbT \underset{1}{\Leftarrow} S$$

Sommaire

Sommaire

Arbre de dérivation

- Un arbre de dérivation indique comment un analyseur reconnaît une phrase
- On l'appelle aussi arbre syntaxique concret.
- Il enregistre la séquence de règles que l'analyseur applique pour reconnaitre une phrase ainsi que les tokens (les terminaux de la grammaire) reconnus.
- Les nœuds internes représentent les variables et les feuilles les terminaux.
- Les arbres de dérivation décrivent des structures en groupant des symboles d'entrée en sous-arbres.

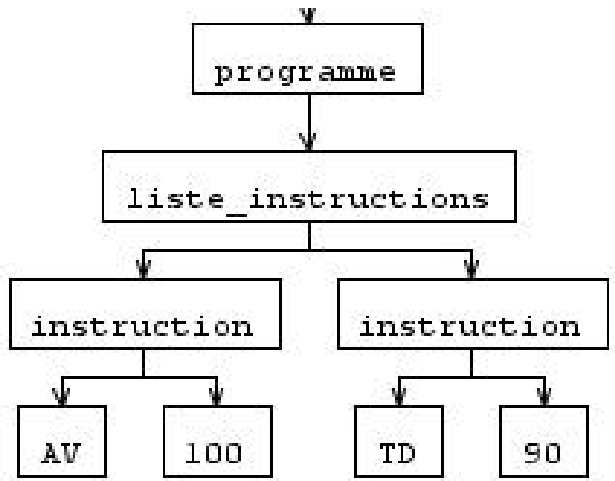
Arbre syntaxique concret

```
programme -> liste_instructions
liste_instructions -> instruction+
instruction -> (AV | TD) INT
```

```
av 100 td 90
```

Le lexer produit :

```
AV INT TD INT
```



Arbre de dérivation - Utilité

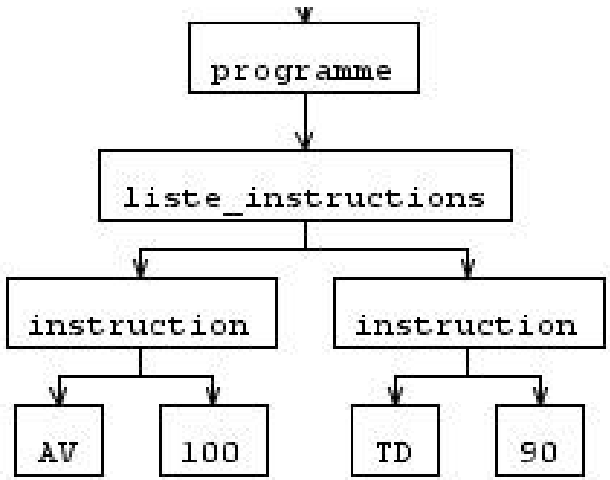
- Pour :
  - facile à construire ;
  - utile pour les environnement de développement (vérification d'erreurs, mise en évidence de la syntaxe) ;
  - utile pour la réécriture de textes (changer le symbole de l'affectation) ou pour des outils d'affichage (afficher la variable dans une affectation).
- Contre :
  - les variables de la grammaire introduisent du bruit ;
  - inutile pour construire un interpréteur ;
  - inutile pour construire un traducteur.

Arbre abstrait

- Un arbre abstrait ne conserve que l'information essentielle de la phrase d'entrée.
- Il ne contient que des terminaux.
- Les nœuds intérieurs sont plutôt des opérateurs.
- Les feuilles sont plutôt des opérandes.
- Les nœuds sont homogènes (convient bien pour les analyseurs en code non-objet comme C).

Comparaison

```
programme -> liste_instructions
liste_instructions -> instruction+
instruction -> (AV | TD) INT
```

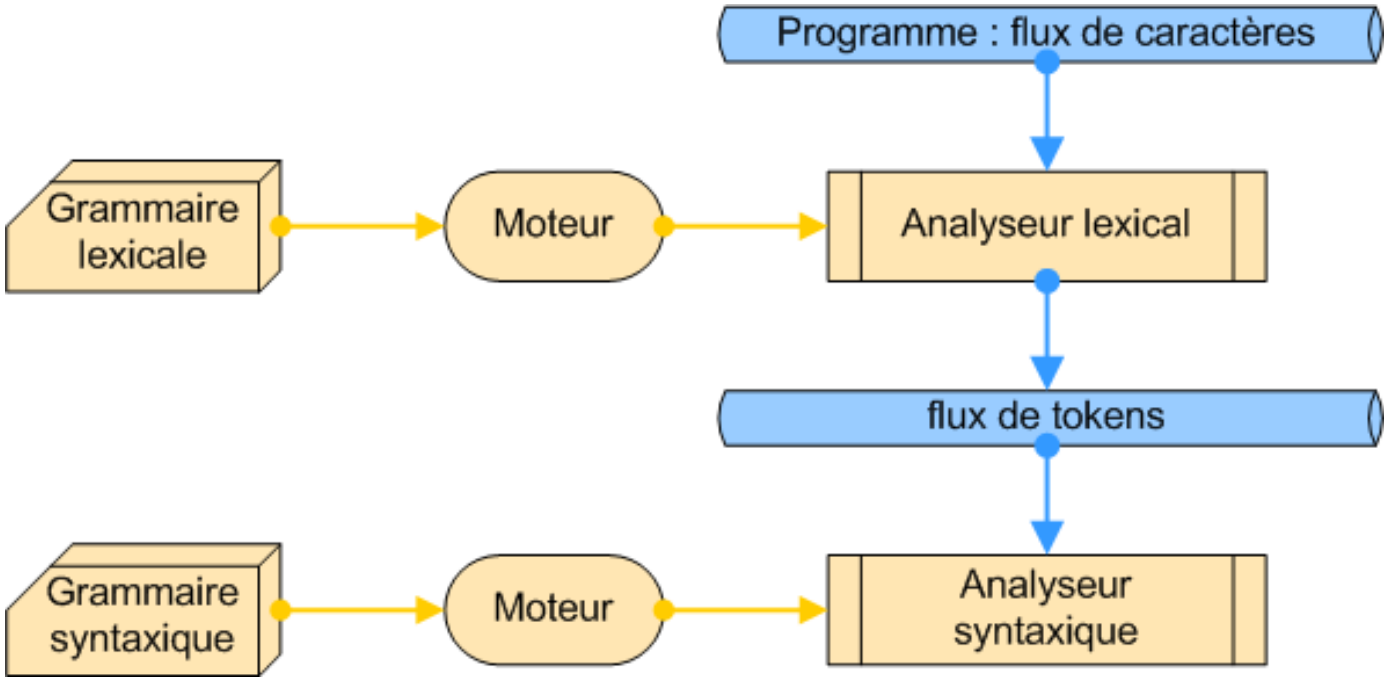



```
av 100 td 90

PROGRAMME
      |___ AV
      |      |___ 100
      |
      |___ TD
             |___ 90
```

Sommaire

Génération : première méthode



- Le parseur généré parcourt l'arbre et doit exécuter des actions.
- On insère dans la grammaire du code représentant les actions à exécuter.

Génération : première méthode

- Sans actions

```
programme -> liste_instructions
liste_instructions -> instruction+
instruction -> (AV | TD) INT
```

- Avec actions

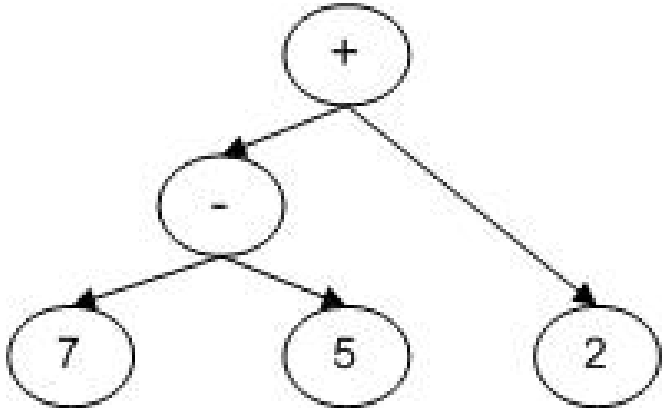
```
programme -> liste_instructions
liste_instructions -> instruction+
instruction ->
      AV INT {System.out.println("Ins. avance")}
    | TD INT
```

Visiteur d'arbre

- Visiter un arbre : exécuter des actions sur les nœuds de l'arbre lors de son parcours.
- 3 parcours possibles :
  - pré-ordre, top-down : visite du nœud parent avant les noeuds fils.
  - ordre intermédiare : visite du nœud parent entre les nœuds fils.
  - post-ordre, bottom-up : visite du nœud parent après les nœuds fils.

Exemple de code pour Visiteur d'arbre

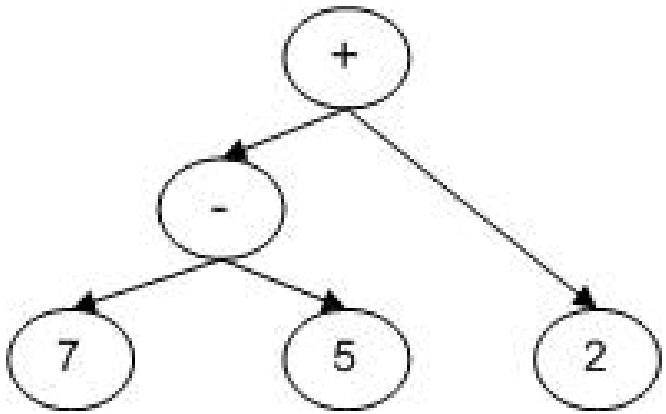
```
public void print(Node nd) {
  // 1 - System.out.print(nd.token);
  if (nd.left != null)
        print(nd.left);
  // 2 - System.out.print(nd.token);
  if (nd.right != null)
        print(nd.right);
  // 3 - System.out.print(nd.token);
}
```



- 1 - pré-ordre : + - 7 5 2
- 2 - intermédiaire : 7 - 5 + 2
- 3 - post-ordre : 7 5 - 2 +

Exemple de code pour Visiteur d'arbre

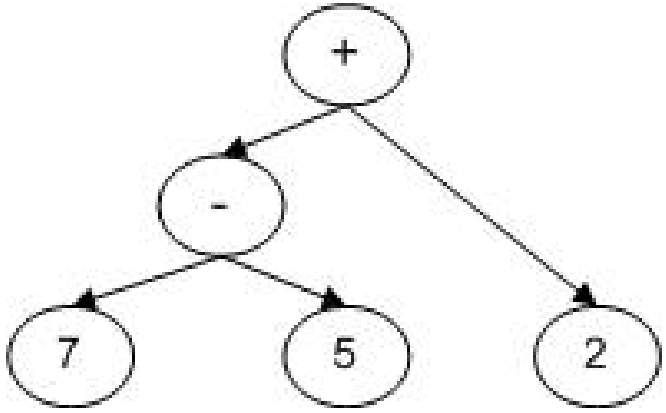
```
public void print(Node nd) {
   System.out.print(nd.token);
   if (nd.left != null)
         print(nd.left);
   if (nd.right != null)
         print(nd.right);
}
```



- 1 - pré-ordre : + - 7 5 2

Exemple de code pour Visiteur d'arbre

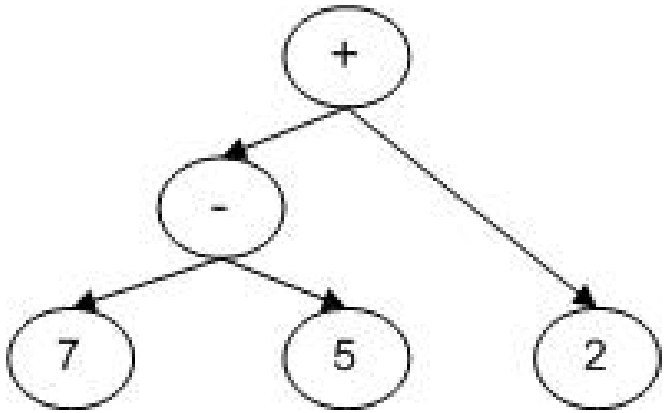
```
public void print(Node nd) {
  if (nd.left != null)
        print(nd.left);
  System.out.print(nd.token);
  if (nd.right != null)
        print(nd.right);
}
```



- 2 - intermédiaire : 7 - 5 + 2

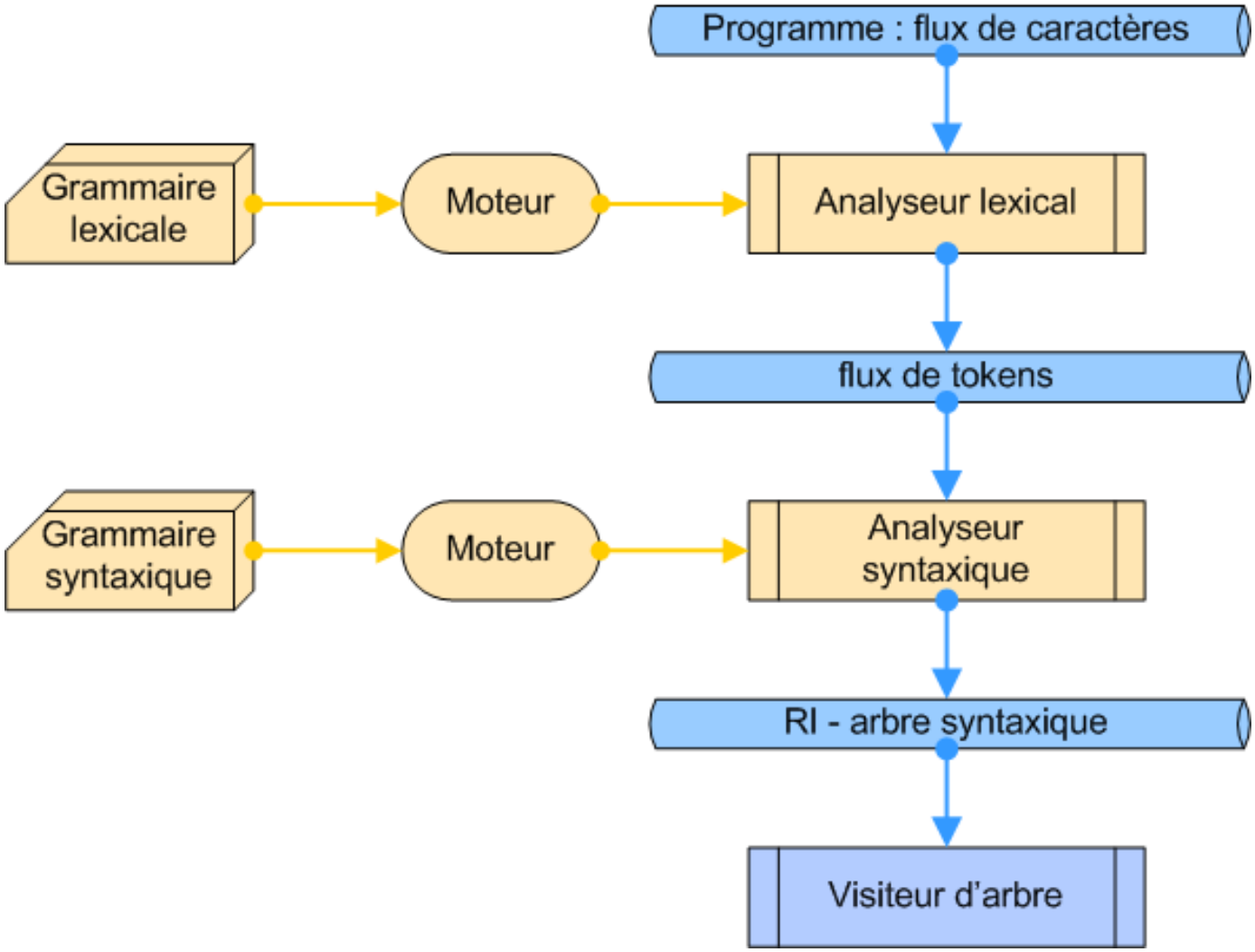
Exemple de code pour Visiteur d'arbre

```
public void print(Node nd) {
  if (nd.left != null)
        print(nd.left);
  if (nd.right != null)
        print(nd.right);
  System.out.print(nd.token);
}
```



- 3 - post-ordre : 7 5 - 2 +

Génération : 2ème méthode



Génération : 2ème méthode

- On laisse intacte la grammaire
- L'analyseur produit un arbre comme représentation intermédiaire d'un programme
- On écrit un visiteur d'arbre avec des méthodes adaptées à chaque type de nœud de l'arbre produit.