PC 10
Analyse Syntaxique

1. Syntaxe et grammaire
2. Analyse descendante
3. Analyse ascendante

Syntaxe

On considère des *langages formels* comme les langages de programmation. On distingue

- La syntaxe: comment on doit écrire.
- La sémantique: ce que ça veut dire.

On utilise pour spécifier la syntaxe des *règles* de grammaire qui ont la forme

symbole → expression

Les expressions utilisent deux sortes de symboles:

1. Ceux du langage final: les symboles *terminaux*
2. Des symboles intermédiaires qui décrivent des *catégories grammaticales*

Définition formelle

Une grammaire est donnée par:

1. Un alphabet A de symboles terminaux.
2. Un ensemble X de symboles non-terminaux.
3. Un ensemble fini de règles
$$(x, w) \in X \times (A + X)^*$$
notées
$$x \to w$$

Un *mot* sur l'alphabet A est une suite d'éléments de A.

On note A^* l'ensemble des mots sur A.

Un langage formel est une partie de A^*. Le langage défini par une grammaire (à partir de x_0) est l'ensemble

$$L = \{w \in A^* \mid x_0 \xrightarrow{*} w\}$$

Exemple

Grammaire pour les expressions arithmétiques simples:

$$
\begin{aligned}
E &\rightarrow nombre \\
E &\rightarrow (\ E\ ) \\
E &\rightarrow E\ +\ E \\
E &\rightarrow E\ -\ E \\
E &\rightarrow E\ *\ E \\
E &\rightarrow E\ /\ E
\end{aligned}
$$

Le symbole E désigne la catégorie grammaticale des expressions arithmétiques. Nous considérons $nombre$ comme un symbole abstrait (destiné à représenter les chaînes de caractères qui sont des nombres).

Les symboles terminaux sont:

$$(\,,\,) \,,\, + \,,\, - \,,\, * \,,\, /$$

Langues naturelles

On peut utiliser les mêmes grammaires pour décrire la syntaxe des langues naturelles. Ainsi on pourra utiliser les règles:

$$
\begin{aligned}
<phrase> &\rightarrow <gsujet><gverbe>\\
<gsujet> &\rightarrow <det><adj><nom>\\
&\quad |<det><nom>\\
&\quad |\ \ldots\\
<gverbe> &\rightarrow <verbe><gcomplement>\\
&\quad |<verbe>\\
&\quad |\ \ldots
\end{aligned}
$$

Ce sont les grammaires de Chomsky.

Arbre d'analyse

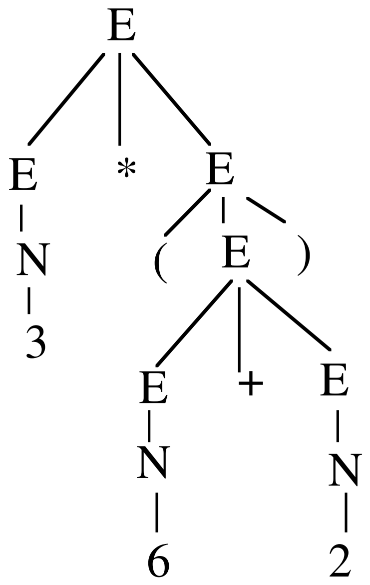

Figure 1: Analyse de $3 * (6 + 2)$

Langage défini par une grammaire: ensemble des frontières d'arbres d'analyse (ce sont des mots sur l'alphabet terminal).

Ambiguité

Grammaire ambigue: un même mot posséde plusieurs arbres d'analyse.

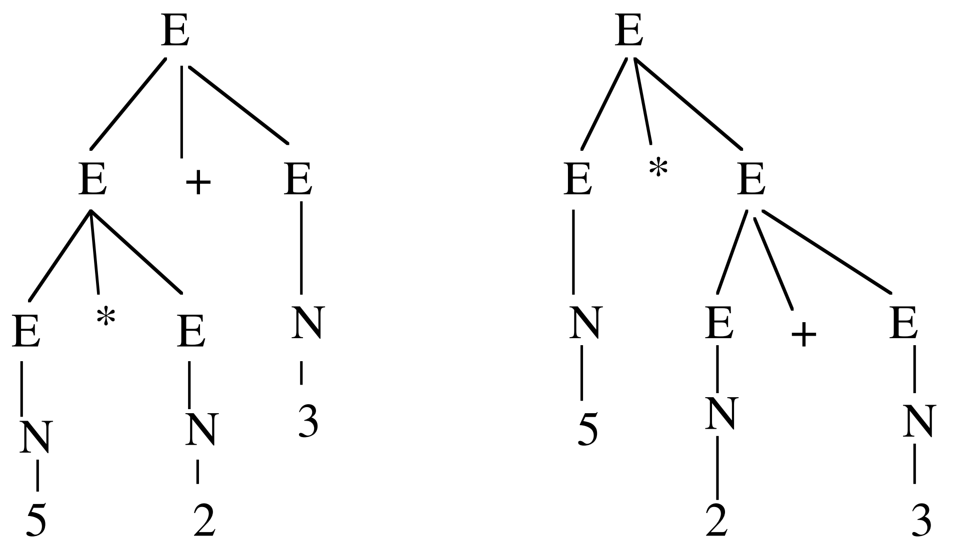

Figure 2: Deux analyses de $5 * 2 + 3$

Exemples

1. La grammaire

$$E \to E + E \mid N$$

   est ambigue.

2. La grammaire

$$P \to (P)P \mid \epsilon$$

   définit les expressions parenthésées. Elle n'est pas ambigue.

3. La grammaire

$$E \to EE+ \mid N$$

   définit les expressions additives en notation suffixe. Elle n'est pas ambigue.

Une grammaire non-ambigue pour les expressions

La grammaire E, T, F: on utilise trois catégories au lieu d'une:

1. Facteur: expression qu'on ne peut pas fragmenter.
2. Terme: produit ou quotient de facteurs.
3. Expression: somme ou différence de termes.

$$
\begin{array}{rcl}
E & \to & E + T \mid E - T \mid T \\
T & \to & T * F \mid T/F \mid F \\
F & \to & (E) \mid N
\end{array}
$$

Choix réalisés:

1. Associativité de gauche à droite.
2. Priorité de $*$ et $/$ sur $+$ et $-$

Analyse syntaxique

Objectif: construction de l'arbre d'analyse d'un mot donné.

Deux méthodes:

- Analyse desendante (=descente recursive): on parcourt la donnée de gauche à droite en appelant des procédures pour chaque symbole de la grammaire.
- Analyse ascendante: on fait le même parcours mais en empilant les expressions identifiées au fur et à mesure.

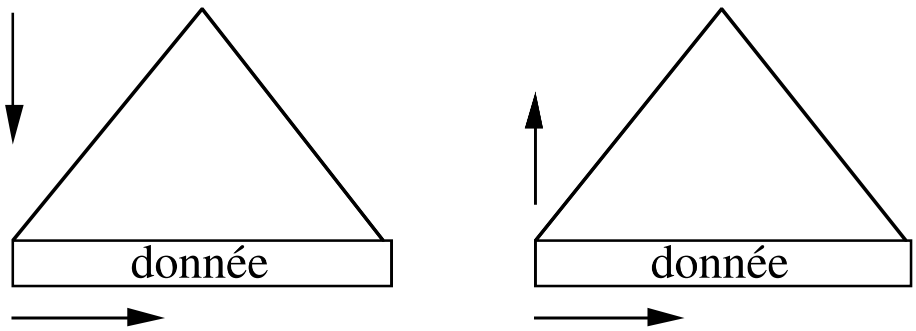

Figure 3: Les deux méthodes

Analyse descendante des expressions arithmétiques

Les procédures Pascal suivantes effectuent la transformation en notation suffixe d'un expression arithmétique. On part de la grammaire:

$$
\begin{array}{rcl}
E & \to & T + E \mid T \\
T & \to & F * T \mid F \\
F & \to & (E) \mid N
\end{array}
$$

On utilise un caractère d'*anticipation* ('look-ahead') pour guider l'analyse. C'est une variable globale de type caractère.

On associe à chaque symbole non-terminal une procédure qui effectue des appels dans l'ordre indiqué par les règles:

```
procedure expression;
begin
 terme;
 if c='+' then begin
  read(c); expression
 end;
write('+')
end;

procedure terme;
begin
 facteur;
 if c='*' then begin
  read(c); terme
 end;
write('*')
end;
```

```
procedure facteur;
begin
 if c='(' then begin
  read(c); expression;
  if c=')' then read(c)
 else write(c)
end;
```

Le programme complet sera:

```
program analyse;

var
 c: char;

prodedure terme; forward;
procedure facteur; forward;
procedure expression;
 ...
procedure terme;
 ...
procedure facteur;
 ...
begin
 read(c);
 expression
end.
```

Premier et Suivant

Pour analyser l'algorithme descendant (algorithme LL) on introduit deux fonctions: Premier et Suivant.

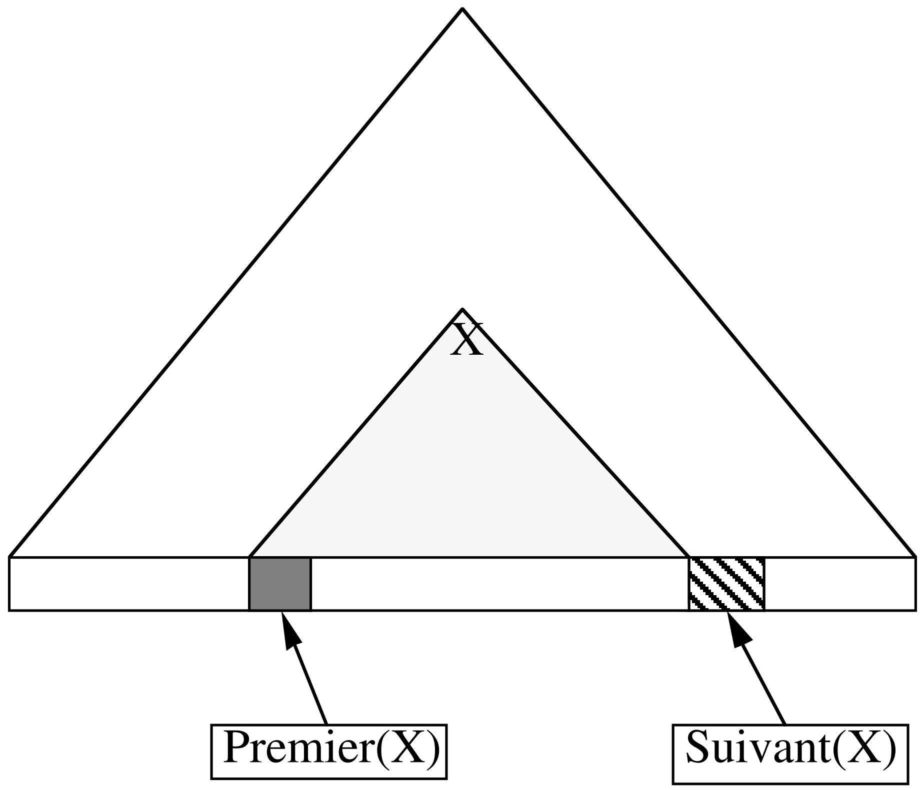

Figure 4:

Calcul de Premier et Suivant

Pour calculer les fonctions Premier et Suivant, on construit un graphe pour chacune. Les sommets sont les symboles terminaux ou non. On a un chemin de X à a ssi $a \in$ Premier(X) (ou Suivant(X)).

Premier

On a une flèche de X vers Y ssi il existe une règle $X \rightarrow UYV$ avec $U \stackrel{*}{\rightarrow} \epsilon$ (Y peut être terminal).

Suivant

Pour calculer Suivant, on aura une flèche de X à Y dans l'un des deux cas suivants:

1. $Y = a$ et il existe une règle $Z \rightarrow UXV$ avec $a \in Premier(V)$
2. il existe une règle $Y \rightarrow UXV$ avec $V \stackrel{*}{\rightarrow} \epsilon$ (attention: on remonte les règles)

Exemples

Pour la grammaire

$$
\begin{array}{rcl}
E & \rightarrow & TE' \\
E' & \rightarrow & +TE' \mid \epsilon \\
T & \rightarrow & FT' \\
T' & \rightarrow & *FT' \mid \epsilon \\
F & \rightarrow & (E) \mid N
\end{array}
$$

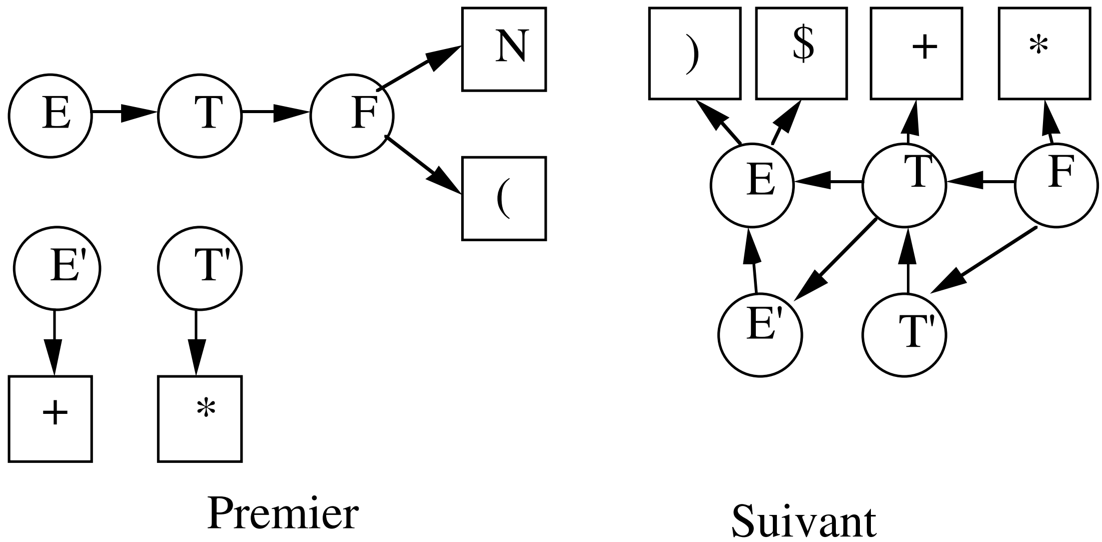

Figure 5:

Table d'analyse LL

On peut utiliser Premier et Suivant pour construire une table d'analyse de la forme:

	...	a	...
X		n	

où la règle n est ou bien

$$X \to U$$

avec $a \in Premier(U)$ ou bien

$$X \to \epsilon$$

avec $a \in Suivant(X)$.

Exemple

	+	*	(	)	N	$
E			1		1	
E'	2			3		3
T			4		4	
T'	6	5		6		6
F			7		8	

$$
\begin{aligned}
1 : E &\rightarrow TE' \\
2 : E' &\rightarrow +TE' \\
3 : E' &\rightarrow \epsilon \\
4 : T &\rightarrow FT' \\
5 : T' &\rightarrow *FT' \\
6 : T' &\rightarrow \epsilon \\
7 : F &\rightarrow (E) \\
8 : F &\rightarrow N
\end{aligned}
$$

Analyse ascendante

On utilise une *pile* pour ranger la partie de la donnée qui a déja été réduite.

Il y a deux sorte de mouvements:

1. Empiler de la donnée vers la pile.
2. Réduire un corps de règle en sommet de pile.

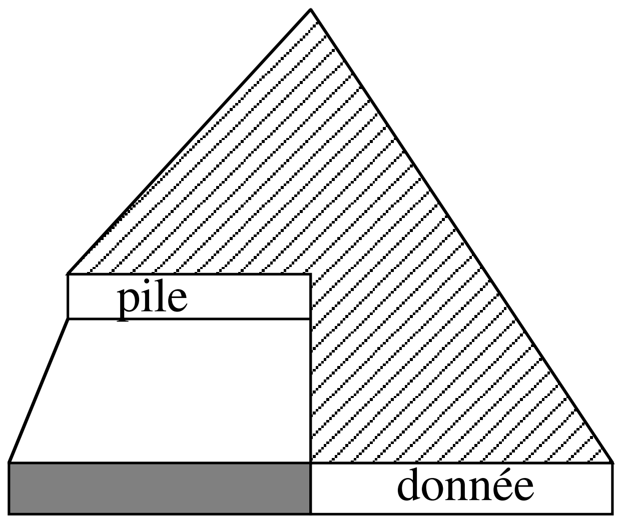

Figure 6: Schéma de fonctionnement

Analyseur LR

On construit un *analyseur LR* utilisant:

1. des *états* donnant l'information sur le contenu de la pile.
2. des *empilements* ajoutant un état en sommet de pile notés ep .
3. des *réductions* par la règle n notées rn
4. des *transitions* donnant l'état après réduction.

ETAT	terminaux	non-terminaux
	a	E
$p : \ldots$	eq	
$q : \ldots$	rn	
$u : \ldots$		v

Si la règle n est $E \to w$, on fait:

1. dépiler $|w|$ symboles.
2. Si on a l'état u au sommet, empiler v.

Analyse SLR(1)

C'est la stratégie la plus simple. Dans la situation:

	...	a	...
p		?	

on ne prend en compte une réduction par la règle $X \to U$ que si $a \in \text{Suivant}(X)$.

Analyse ascendante des expressions arithmétiques

On prend la grammaire:

$$\begin{aligned}
1: E &\rightarrow E + T \\
2: E &\rightarrow T \\
3: T &\rightarrow T * F \\
4: T &\rightarrow F \\
5: F &\rightarrow (E) \\
6: F &\rightarrow N
\end{aligned}$$

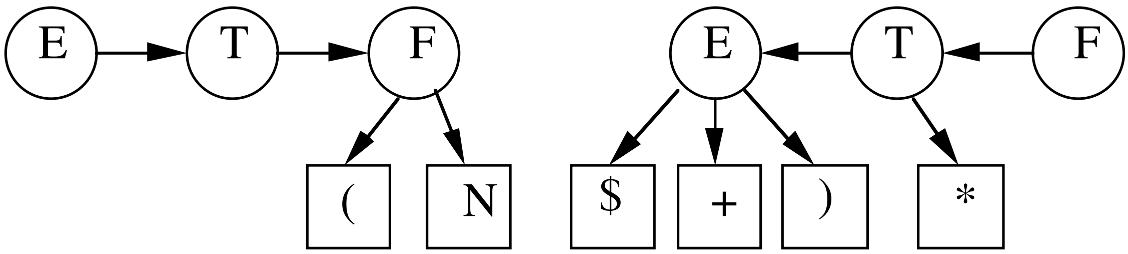

Figure 7: Premier et Suivant

La table d'analyse

ETAT	terminaux						non-termin.		
	N	+	$*$	(	)	\$	E	T	F
0	e5			e4			1	2	3
1: E		e6				acc			
2: T ou $\ldots(T$		r2	**e7**		r2	r2			
3: $\ldots F$		r4	r4		r4	r4			
4: $\ldots($	e5			e4			8	2	3
5: $\ldots N$		r6	r6		r6	r6			
6:$\ldots E+$	e5			e4				9	3
7: $\ldots T*$	e5			e4					10
8: $\ldots(E$		e6			e11				
9: $\ldots E+T$		r1	**e7**		r1	r1			
10: $\ldots T*F$		r3	r3		r3	r3			
11: $\ldots(E)$		r5	r5		r5	r5			

Calcul des états

Les états sont des ensembles de *règles marquées*

$$x \to u.v$$

correspondant à la situation: on a déja u en sommet de pile et on attend d'empiler v pour réduire uv en x.

Les transitions sont de deux sortes:

1. empilement: passer de $x \to u.yv$ à $x \to uy.v$ avec lecture de y.
2. réduction: passer de $x \to u.yv$ à $y \to w$ sans lecture de symbole.

$$0: \begin{array}{l} E \rightarrow .E + T \\ E \rightarrow .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .N \end{array} \qquad 1: E \rightarrow E. + T \quad 2: E \rightarrow T.$$

$$3: T \rightarrow F. \qquad 4: F \rightarrow (.E) \qquad 5: F \rightarrow N.$$

$$6: \begin{array}{l} E \rightarrow E + .T \\ T \rightarrow .T * F \\ T \rightarrow .F \\ F \rightarrow .(E) \\ F \rightarrow .N \end{array}$$

L'automate LR

Analyse $LR(1)$

On prend cette fois dans l'état la lettre attendue comme lookahead au moment de la réduction. Les états sont des ensembles de couples

$$(x \to u.v, a)$$

avec les deux sortes de transitions:

1. empilement: sans changer de lookahead
2. réduction: on passe de $(x \to u.yv, a)$ à $(y \to w, b)$ avec $b \in \text{Premier}(va)$

Utilisation de grammaires ambigues

On peut résoudre les conflits de l'analyse LR, même sur une grammaire ambigue en utilisant des règles de priorité. Ainsi, pour la grammaire:

$$
\begin{array}{lcl}
E & \rightarrow & E + E \\
E & \rightarrow & E * E \\
E & \rightarrow & (E) \\
E & \rightarrow & \mathbf{id}
\end{array}
$$

on obtient une table d'analyse LR en résolvant les conflits par le choix de l'associativité de gauche à droite et la priorité de $*$ sur $+$.

Automate LR

ETAT	terminaux						non-termin.
	id	+	$*$	(	)	\$	E
0	e3			e2			1
1		e4	e5			ok	
2	e3			e2			6
3		r4	r4		r4	r4	
4	e3			e2			8
5	e3			e2			8
6		e4	e5		e9		
7		r1	e5		r1	r1	
8		r2	r2		r2	r2	
9		r3	r3		r3	r3	

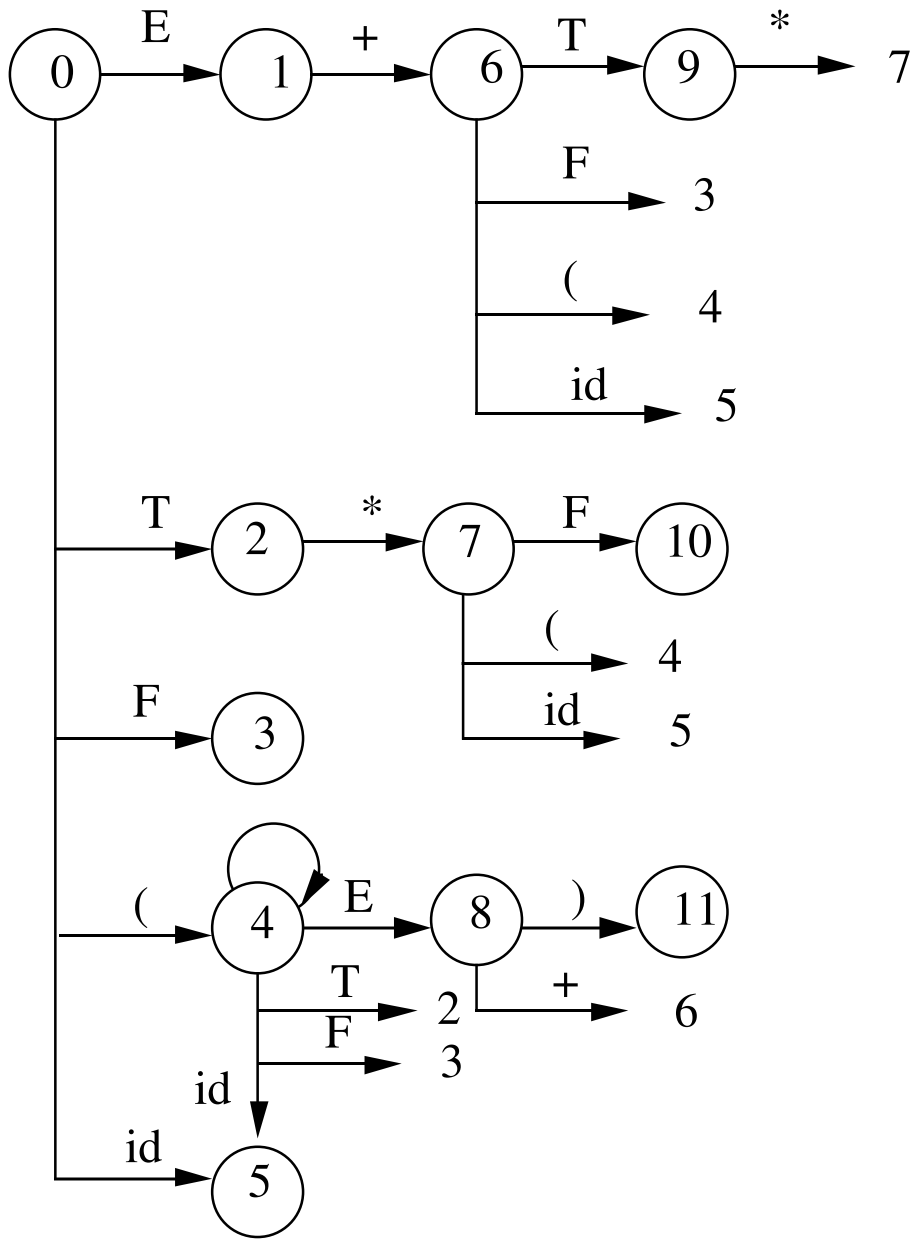

Figure 8:

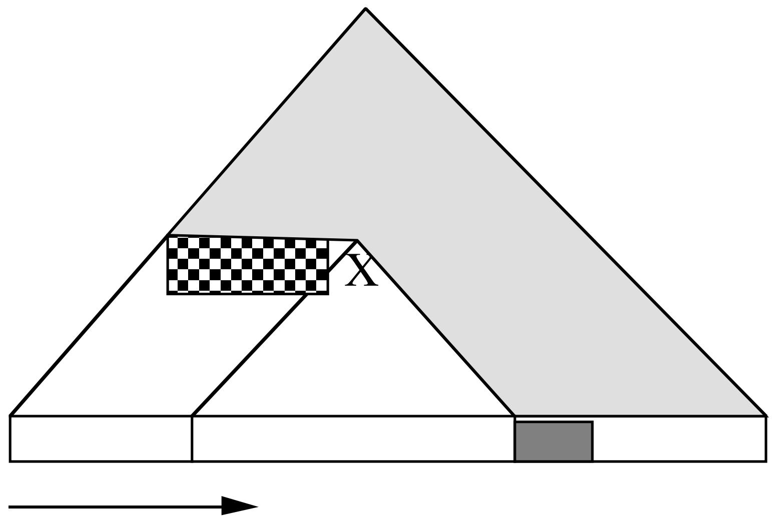

Figure 9:

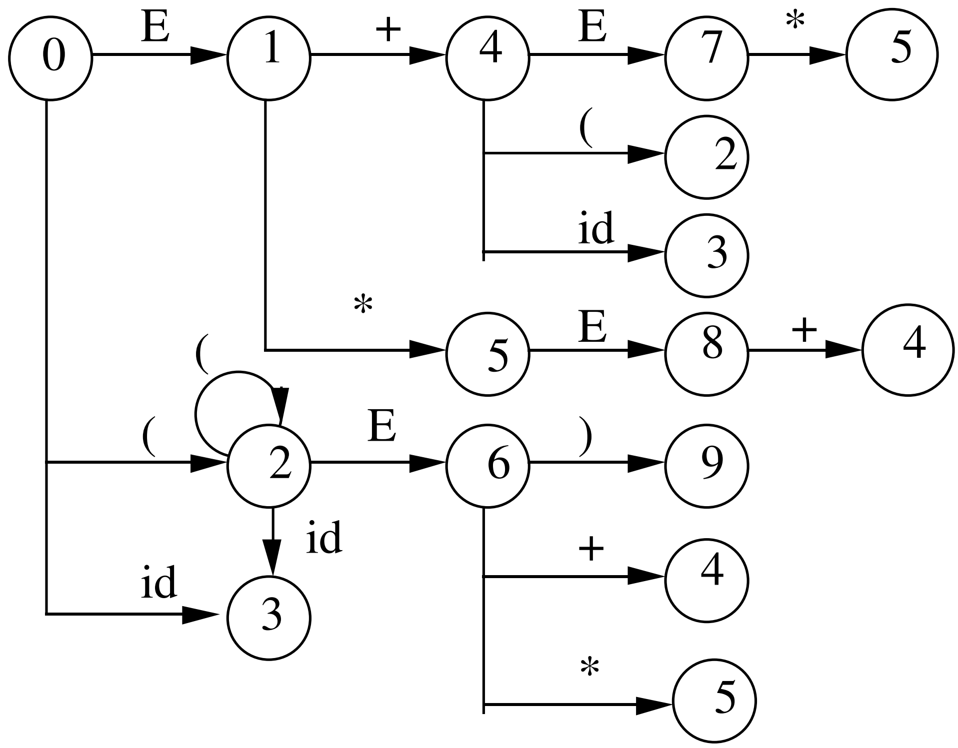

Figure 10: