Chapitre 03 : Analyse syntaxique

1. Introduction
2. Grammaires hors contexte
3. Arbre de dérivation
4. Ambiguïté
5. Mise en œuvre d'un analyseur syntaxique
6. Analyse descendante: Analyseur LL(1)
7. Analyse ascendante
8. Outil Yacc (générateur d'analyseur syntaxique)

Introduction

Tout langage de programmation possède des règles qui indiquent la structure syntaxique d'un programme bien formé. Par exemple, en Pascal, un programme bien formé est composé de blocs, un bloc est formé d'instructions, une instruction de ... . La **syntaxe** d'un langage peut être décrite par une **grammaire**.
L'analyseur syntaxique reçoit une suite d'unités lexicales de la part de l'analyseur lexical et doit vérifier que cette suite peut être engendrée par la grammaire du langage.
➔Est ce que *m* appartient au langage généré par *G* ? Le principe est d'essayer de construire un **arbre de dérivation**.

Grammaires hors contexte

Une grammaire est un ensemble de règles décrivant comment former des phrases.

Une **grammaire** est la donnée de $G=(V_T,V_N,S,P)$ où

- V_T est un ensemble non vide de symboles ***terminaux*** (alphabet terminal)
- V_N est un ensemble de symboles ***non-terminaux***, avec $\mathbf{V_T} \cap \mathbf{V_N}$
- S est un symbole initial $\in \mathbf{V_N}$ appelé ***axiome***
- P est un ensemble de ***règle de productions***.

Arbre de dérivation

On appelle ***arbre de dérivation*** (ou arbre syntaxique) tout arbre tel que

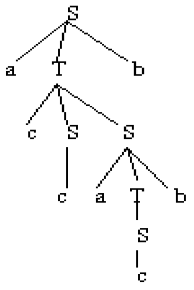
- la racine est l'axiome
- les feuilles sont des unités lexicales ou ε
- les nœuds sont des symboles non-terminaux
- les fils d'un nœud α sont $\beta_0 \ldots \beta_n$ si et seulement si $\alpha \rightarrow \beta_0 \ldots \beta_n$ est une production (Le parcours préfixe de l'arbre donne le mot)

Exemple

Soit la grammaire ayant *S* pour axiome et pour règles de production

$$P = \begin{cases} S \rightarrow aTb \mid c \\ T \rightarrow cSS \mid S \end{cases}$$

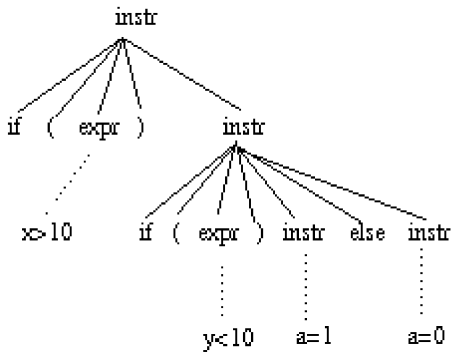
Un arbre de dérivation pour le mot ***accacbb*** est :

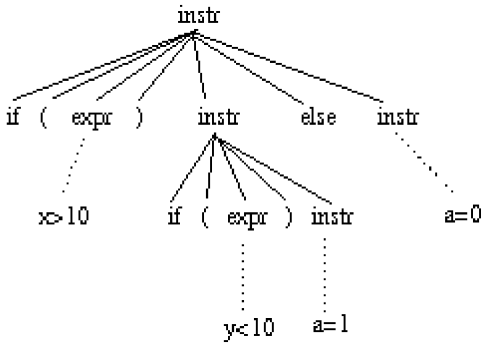


$S \rightarrow aTb \rightarrow acSSb \rightarrow accSb \rightarrow accaTbb \rightarrow accaSbb \rightarrow accacbb$ (dérivations **gauches**)
ou $S \rightarrow aTb \rightarrow acSSb \rightarrow acSaTbb \rightarrow acSaSbb \rightarrow acSacbb \rightarrow accacbb$ (dérivations **droites**)
Ces deux suites **différentes** de dérivations donnent le **même** arbre de dérivation.

Ambiguïté

Une grammaire est dite ***ambiguë*** s'il existe un mot de $L(G)$ ayant plusieurs arbres syntaxiques.
Exemple de grammaire ambiguë : Soit *G* donnée par:
instr→ if (expr) instr else instr
inst→ if (expr) instr if (expr) instr
inst→ …
expr→…
Cette grammaire est ambiguë car le mot ***m*= if (*x*>10) if (*y*<0) *a*=1 else *a*=0** possède deux arbres syntaxiques différents :





Car il y a deux interprétations syntaxiques possibles :

```
if (x>10)                if (x>10)
   if (y<0)                 if (y<0)
      a=1                      a=1
   else                     // finsi
      a=0                else
   // finsi                    a=0
// finsi                 // finsi
```

Solution proposée

stmt → matched-stmt
| open-stmt
matched-stmt→ if expr then matched-stmt else matched-stmt
| other
open-stmt → if expr then stmt
| if expr then matched-stmt else open-stmt

7. Mise en œuvre d'un analyseur syntaxique

Il existe deux approches: une méthode descendante et une méthode ascendante.

8. Analyse descendante: Analyseur LL(1)

Récursivité à gauche

Une grammaire **G** est **Récursive Gauche** (**RG**) s'il existe une dérivation de la forme : $A \rightarrow^{+} Aw$, avec $A \in \mathbf{N}$ et $w \in (\mathbf{N} \cup \mathbf{T})^*$
Une récursivité $A \rightarrow Aw$ est dite **immédiate**
Exemple :

$$\begin{cases} S \rightarrow ScA|B \\ A \rightarrow Aa|\varepsilon \\ B \rightarrow Bb|d|e \end{cases}$$

Cette grammaire contient plusieurs récursivités à gauche immédiates.

Elimination de la récursivité à gauche immédiate

Remplacer toute règle de la forme **A→Aα|β** par les deux règles :
A→βA'
A'→ αA'| Є

Elimination de la récursivité à gauche

Ordonner les non-terminaux $A_1, A_2, \ldots A_n$
Pour i=1 à n faire
pour j=1 à i-1 faire
remplacer chaque production de la forme $\mathbf{A_i \rightarrow A_j\alpha}$ **où** $\mathbf{A_j \rightarrow \beta_1|\ldots|\beta_p}$ par $\mathbf{A_j \rightarrow \beta_1\alpha|\ldots|\beta_p\alpha}$
fin pour
éliminer les récursivités à gauche immédiates des productions A_i
fin pour

Exemple

Soit la grammaire suivante:
S→Aa|b
A→Ac|Sd|BA|c
B→SSc|a

On ordonne ***S,A,B*** :
S→Aa|b
i=1 pas de récursivité immédiate dans **S→Aa|b**
i=2 et j=1 on obtient
A→Ac|Aad|bd|BA|c

on élimine la récursivité immédiate :

A→bdA'|BAA'|cA'
A'→ cA'|bdA'|Є

i=3 et j=1 on obtient **B→AaSc|bSc|a**
et j=2 donne **B→ bdA'aSc| BAA'aSc| cA'aSc|bSc|a**
on élimine la récursivité immédiate:

B→ bdA'aScB' | cA'aScB' |bScB' |a B'
B'→ AA'aScB' | Є

On a obtenu:

S→Aa|b
A→bdA'|BAA'|cA'
A'→ cA'|bdA'|Є
B→ bdA'aScB' | cA'aScB'|bScB'|aB'
B'→ AA'aScB'| Є

Factorisation à gauche

L'idée de base est que pour développer un non-terminal *A* quand il n'est pas évident de choisir l'alternative à utiliser (c à d quelle production prendre), on doit réécrire les productions de *A* de façon à **différer** la décision jusqu'à ce que suffisamment de texte ait été lu pour faire le bon choix.

Exemple

$$\begin{cases} S \rightarrow bacdAbd | bacdBcca \\ A \rightarrow aD \\ B \rightarrow cE \\ C \rightarrow \dots \\ E \rightarrow \dots \end{cases}$$

Au départ, pour savoir s'il faut choisir $S \rightarrow bacdAbd$ ou $S \rightarrow bacdBcca$, il faut avoir lu la 5ème lettre du mot (un a ou un c). On ne peut donc pas dès le départ savoir quelle production prendre.

Factorisation à gauche

Pour chaque non-terminal *A*
trouver le plus long préfixe α commun à deux de ses alternatives ou plus
Si $\alpha \neq \varepsilon$, remplacer $A \rightarrow \alpha\beta_1|\dots|\alpha\beta_n|\gamma_1|..|\gamma_p$ (où les γ_i ne commencent pas par α) par le deux règles:

$A \rightarrow \alpha A'|\gamma_1|..|\gamma_p$
$A' \rightarrow \beta_1|\dots|\beta_n$

finpour
Recommencer jusqu'à ne plus en trouver.

Exemple

S→aEbS|aEbSeB|a
E→bcB|bca
B→ba

Factorisée à gauche, cette grammaire devient :
S→aS''
S''→EbSS'|Є
S'→eB| Є
E →bcE'
E'→B|a
B→ba

Table d'analyse LL(1)

Pour construire une table d'analyse, on a besoin des ensembles PREMIER et SUIVANT

Calcul de PREMIER

Pour toute chaîne $\alpha \in (V_t \cup V_n)^*$, on cherche PREMIER($\alpha$): l'ensemble de tous les **terminaux** qui peuvent **commencer** une chaîne qui se dérive de α, C'est à dire que l'on cherche toutes les lettres *a* telles qu'il existe une dérivation $\alpha \rightarrow a\beta$ ($\beta\ \alpha \in (V_t \cup V_n)^*$,).

Algorithme de construction des ensembles PREMIER

1. Si *X* est un non-terminal et $X \rightarrow Y_1, Y_2 \ldots Y_n$ est une production de la grammaire (avec Y_i symbole terminal ou non-terminal) alors:
   - ajouter les éléments de PREMIER(Y_1) **sauf** ε dans PREMIER(*X*)
   - s'il existe j ($j \in \{2,\ldots,n\}$) tel que pour tout i=1,…,j-1 on a $\varepsilon \in$ PREMIER(Y_i), alors ajouter les éléments de PREMIER(Y_j) **sauf** ε dans PREMIER(X)
   - si pour tout i=1,…, n, $\varepsilon \in$ PREMIER(Yi), alors ajouter ε dans PREMIER(X)
2. Si X est un non-terminal et $X \rightarrow \varepsilon$ est une production, ajouter ε dans PREMIER(X)
3. Si X est un terminal, PREMIER(X)={X}
4. Recommencer jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles PREMIER.

Exemple

E→TE'
E'→+TE'|Є
T→FT'
T'→*FT'|Є
F→(E)|nb

PREMIER(E) = PREMIER(T)={(,nb}
PREMIER(E') ={+,Є}
PREMIER(T) = PREMIER(F) ={(,nb}
PREMIER(T') ={*, Є }
PREMIER(F) ={(,nb}

Calcul de SUIVANT

Pour tout non-terminal A, on cherche SUIVANT(*A*) : l'ensemble de tous les symboles terminaux *a* qui peuvent apparaître immédiatement à droite de *A* dans une dérivation : $S \rightarrow^* \alpha A a \beta$

Algorithme de construction des ensembles SUIVANT :

1. Ajouter un marqueur de fin de chaîne (symbole $ par exemple) à SUIVANT(*S*) (*S* est l'axiome)
2. S'il y a une production **A→αBβ** où *B* est un non-terminal, alors ajouter le contenu de PREMIER(β) à SUIVANT(*B*), **sauf ε**
3. S'il y a une production **A→αB**, alors ajouter SUIVANT(*A*) à SUIVANT(*B*)
4. S'il y a une production **A→αBβ** avec $\varepsilon \in$ PREMIER(β), alors ajouter SUIVANT(*A*) à SUIVANT(*B*).
5. Recommencer à partir de l'étape 3 jusqu'à ce qu'on n'ajoute rien de nouveau dans les ensembles SUIVANT.

Exemple:

1. E→TE'	**Suivant(E)={ $,)}**
2. E'→+TE'|Є	**Suivant(E')={$,)}**
3. T→FT'	**Suivant(T)={+,),$}**
4. T'→*FT'|Є	**Suivant(T')={ +,),$}**
5. F→(E)|nb	**Suivant(F)={ *,+,),$}**

Construction de la table d'analyse

Une table d'analyse est un tableau M à deux dimensions qui indique pour chaque symbole non-terminal *A* et chaque symbole terminal *a* ou symbole $ la règle de production à appliquer.

- Pour chaque production $A \rightarrow \alpha$ faire
  1. pour tout a ∈ PREMIER(α) (et $a \neq \varepsilon$), rajouter la production **A→α** dans la case M[A,a]
  2. si ε ∈ PREMIER(α), alors pour chaque b ∈ SUIVANT(*A*) ajouter **A→α** dans *M*[*A*,*b*]
- Chaque case M[A,a] vide est une erreur de syntaxe

Avec l'exemple 1, on obtient la table

	nb	**+**	*****	**(**	**)**	**$**
E	**E→TE'**			**E→TE'**		
E'		**E'→+TE'**			**E'→Є**	**E'→Є**
T	**T→FT'**			**T→FT'**		
T'		**T'→Є**	**T'→*FT'**		**T'→Є**	**T'→Є**
F	**F→nb**			**F→(E)**		

Analyseur syntaxique

Comment utiliser la table pour déterminer si un mot *m* donné est tel que **S→*m?** On utilise une **pile**.
Algorithme :
Données : mot *m*, table d'analyse *M* et un pointeur ***ps*** sur la 1ère lettre de *m*

Initialisation de la pile :

S
$

Répéter
  Soit *X* le symbole en sommet de pile
  Soit *a* la lettre pointée par *ps*
  Si *X* est un non terminal **alors**
    Si M[A,a]= X→Y1…Yn alors
      enlever *X* de la pile
      mettre Y_n puis Y_{n-1} puis ...puis Y_1 dans la pile
      émettre en sortie la production $X \rightarrow Y_1 \ldots Y_n$
    sinon
      ERREUR
    finsi
  Sinon
    Si *X*=$ **alors**
      Si *a*=$ **alors** ACCEPTER
      Sinon ERREUR
      Finsi
    Sinon
      Si *X*=*a* **alors**
        enlever *X* de la pile
        avancer *ps*
      Sinon
        ERREUR
      finsi
    finsi
  finsi
jusqu'à ERREUR ou ACCEPTER

Sur notre exemple (grammaire *E*,*E*',*T*,*T*',*F*), soit le mot *m*=3+4*5

PILE	Entrée	Sortie
\$ *E*	3+4*5\$	**E→TE'**
\$ *E'T*	3+4*5\$	**T→FT'**
\$ *E'T'F*	3+4*5\$	**F→nb**
\$ *E'T'*3	3+4*5\$	
\$ *E'T'*	+4*5\$	**T'→Є**
\$ *E'*	+4*5\$	**E'→+TE'**
\$ *E'T*+	+4*5\$	
\$ *E'T*	4*5\$	**T→FT'**
\$ *E'T'F*	4*5\$	**F→nb**
\$ *E'T'*4	4*5\$	
\$ *E'T'*	*5\$	**T'→*FT'**
\$ *E'T'F**	*5\$	
\$ *E'T'F*	5\$	**F→nb**
\$ *E'T'*5	5\$	
\$ *E'T'*	\$	**T'→Є**
\$ *E'*	\$	**E'→Є**
\$	\$	**analyse syntaxique réussie**

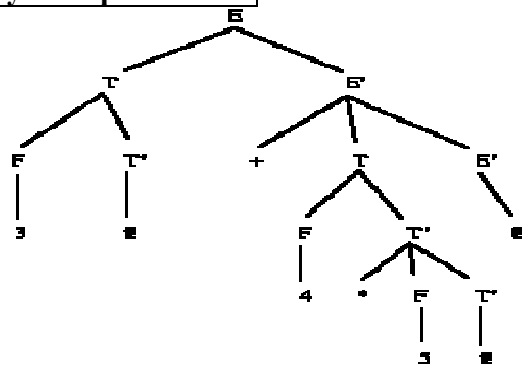


Figure 5.1: Arbre syntaxique pour 3+4*5

Si l'on essaye d'analyser maintenant le mot ***m*=(7+3)5**

PILE	Entrée	Sortie
\$ *E*	(7+3)5\$	**E→TE'**
\$ *E'T*	(7+3)5\$	**T→FT'**
\$ *E'T'F*	(7+3)5\$	**F→(nb)**
\$ *E'T'*) *E* (	(7+3)5\$	
\$ *E'T'*) *E*	7+3)5\$	**E→TE'**
\$ *E'T'*)*E'T*	7+3)5\$	**T→FT'**
\$ *E'T'*)*E'T'F*	7+3)5\$	**F→7**
\$ *E'T'*)*E'T'*7	7+3)5\$	
\$ *E'T'*)*E'T'*	+3)5\$	**T'→Є**
\$ *E'T'*)*E'*	+3)5\$	**E'→+TE'**
\$ *E'T'*)*E'T*+	+3)5\$	
\$ *E'T'*)*E'T*	3)5\$	**T→FT'**
\$ *E'T'*)*E'T'F*	3)5\$	**F→3**
\$ *E'T'*)*E'T'*3	3)5\$	
\$ *E'T'*)*E'T'*	)5\$	**T'→Є**
\$ *E'T'*)*E'*	)5\$	**E'→ Є**
\$ *E'T'*)	)5\$	

$ *E'T*	5$	**ERREUR !!**

Donc ce mot n'appartient pas au langage généré par cette grammaire.

9. Analyse ascendante

Construire un arbre de dérivation du bas (les feuilles, ou les unités lexicales) vers le haut (la racine, ou l'axiome de départ).
Le modèle général utilisé est le modèle par **décalages–réductions**. C'est à dire que l'on ne s'autorise que deux opérations :
Décalage (shift) : décaler d'une lettre le pointeur sur le mot en entrée
Réduction (reduce) : réduire une chaîne (suite consécutive de terminaux et non terminaux à gauche du pointeur sur le mot en entrée et finissant sur ce pointeur) par un non-terminal en utilisant une des règles de production

Table d'analyse LR

- C'est une sorte d'automate, qu'on appelle *automate à pile*. Cette table va nous dire ce qu'il faut faire quand on lit une lettre ***a*** et qu'on est dans un état ***i***.
- Soit on **décale**. Dans ce cas, on empile la lettre lue et on va dans un autre état *j*. On note ça **d*j***
- Soit on **réduit** par la règle de production numéro ***p***, c à d qu'on remplace la chaîne en sommet de pile par le non-terminal de la partie gauche de la règle de production, et on va dans l'état *j* qui dépend du non-terminal en question. On note ça **r*p***.
- Soit on **accepte** le mot. Ce qui sera noté ACC.
- Soit c'est une **erreur**. c à d Case vide.

Construction de la table d'analyse

Utilise aussi les ensembles **SUIVANT** et **PREMIER**, plus ce qu'on appelle des fermetures de **0-items.** Un 0-item (ou *item*) est une production de la grammaire avec un "**.**" quelque part dans la partie droite.
Par exemple :
E → E . + T ou encore T→ F. ou encore F → . (E)

Fermeture d'un ensemble d'items *I* :

1- Mettre chaque item de *I* dans Fermeture(*I*)
2- Pour chaque item *i* de Fermeture(*I*) de la forme **A→α. *B* β**
  Pour chaque production $B \rightarrow \gamma_i$
    rajouter l'item $B \rightarrow .\gamma_i$ dans Fermeture (*I*)
3- Recommencer 2 jusqu'à ce qu'on n'ajoute rien de nouveau

Exemple
Soit la grammaire des expressions arithmétiques

1. **E → E+ T**
2. **E → T**
3. **T →T* F**
4. **T → F**
5. **F → (E)**
6. **F → n**

Et soit l'ensemble d'items **{ T →T * . F, E→E.+T}.** La fermeture de cet ensemble d'items est :
{T→T*.F, E→E.+T, F→.nb, F→.(E)}

Transition par X d'un ensemble d'items I :

$\Delta(I,X)$=Fermeture (tous les items A→α*X* . β où A→α. *Xβ* est dans *I*)

Sur l'exemple ETF : soit I={T→T*.F, E→E.+T, F→.nb, F→.(E)} on aura
$\Delta(I,F)$ = *{ T→T*F.}*
$\Delta(I,+)$ = *{ E →E+.T, T→.T*F, T→.F, F→.*nb,*F→.(E)}*

Collection des items d'une grammaire :

0- Rajouter l'axiome S' avec la production $S' \rightarrow S$
1- Mettre dans l'item I_0 la Fermeture de $S' \rightarrow .S\})$
2- Mettre I_0 dans Collection
3 - Pour chaque I dans Collection faire
  Pour chaque X tel que $\Delta(I, X)$ est non vide
    ajouter ce $\Delta(I, X)$ dans Collection
  Fin pour
4 - Recommencer 3 jusqu'à ce qu'on n'ajoute rien de nouveau

Exemple

I_0={S→.E, E →.E+T, E →.T, T→.T*F, T→. F, F →.nb, F →.(E) }
$\Delta(I_0,E)$= {S→E., E →E.+T}=I_1
$\Delta(I_0,T)$= {E → T., T→T.*F}=I_2
$\Delta(I_0,F)$= {T→ F.}=I_3
$\Delta(I_0,()$= {F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_0,nb)$={ F →nb.}=I_5
$\Delta(I_1,+)$= **{E →E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=I_6**
$\Delta(I_2,*)$={ T→T*.F, F →.nb, F →.(E)}=I_7
$\Delta(I_4, E)$= {F→(E.), E→E.+T}=I_8
$\Delta(I_4, T)$={ E→T., T→T.*F }=I_2
$\Delta(I_4, F)$={ T→F. }=I_3
$\Delta(I_4,nb)$={ F→nb. }=I_5
$\Delta(I_4,()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_6, T)$= {E →E+T., T→T.*F }=I_9
$\Delta(I_6, F)$={ T→F. }=I_3
$\Delta(I_6, nb)$={ F→nb. }=I_5
$\Delta(I_6, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_7, F)$={ T→T*F.}=I_{10}
$\Delta(I_7, nb)$={ F→nb. }=I_5
$\Delta(I_7, ()$={ F→(.E), E→.E+T, E→.T, T→.T*F, T→.F, F→.nb, F→.(E)}=I_4
$\Delta(I_8,))$= {F→(E).}=I_{11}
$\Delta(I_8, +)$= {F→E+.T, T→.T*F, T→. F, F →.nb, F →.(E) }=I_6
$\Delta(I_9,*)$={ T→T*.F, F →.nb, F →.(E)}=I_7

Construction de la table d'analyse SLR :

1- Construire la collection d'items {I_0, ... I_n}
2- l'état i est construit à partir de I_i :
  a) pour chaque $\Delta(I_i , a) = I_j$: mettre **décalage par** j dans la case M[i,a]
  b) pour chaque $\Delta(I_i , A) = I_j$: mettre **aller à** j dans la case M[i,A]
  c) pour chaque $A \rightarrow \alpha.$ contenu dans I_i :
    pour chaque a de SUIVANT(A) faire
      mettre **réduction par** numéro (de la règle $A \rightarrow \alpha$) dans la case M[i,a]

Toujours le même exemple : il nous faut les SUIVANT et PREMIER :

	PREMIER	**SUIVANT**
E	**nb (**	**$ +)**
T	**nb (**	**$ + *)**
F	**nb (**	**$ + *)**

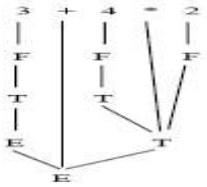
Avec notre exemple ETF, on obtient la table d'analyse LR

état	nb	+	*	(	)	$	E	T	F
0	d5			d4			1	2	3
1		d6				ACC			
2		r2	d7		r2	r2			
3		r4	r4		r4	r4			
4	d5			d4			8	2	3
5		r6	r6		r6	r6			
6	d5			d4				9	3
7	d5			d4					10
8		d6			d11				
9		r1	d7		r1	r1			
10		r3	r3		r3	r3			
11		r5	r5		r5	r5			

Analyseur syntaxique SLR :

On part dans l'état 0, et on empile et dépile non seulement les symboles (comme lors de l'analyseur LL) mais aussi les états successifs.

Exemple :
L'analyse du mot m=3+*4$ est donnée figure 5.3. La figure 5.4 donne l'analyse du mot 3+4*2$. On peut aussi dessiner l'arbre obtenu (figure 5.5)



pile	entrée	action
$ 0	$3 + 4 * 2\$$	d5
$ 0 3 5	$+4 * 2\$$	r6 : $F \to$ nb
$ 0 F	$+4 * 2\$$	je suis en 0 avec F : je vais en 3
$ 0 F 3	$+4 * 2\$$	r4 : $T \to F$
$ 0 T	$+4 * 2\$$	je suis en 0 avec T : je vais en 2
$ 0 T 2	$+4 * 2\$$	r2 : $E \to T$
$ 0 E	$+4 * 2\$$	je suis en 0 avec E : je vais en 1
$ 0 E 1	$+4 * 2\$$	d6
$ 0 E 1 + 6	$4 * 2\$$	d5
$ 0 E 1 + 6 4 5	$*2\$$	r6 : $F \to$ nb
$ 0 E 1 + 6 F	$*2\$$	je suis en 6 avec F : je vais en 3
$ 0 E 1 + 6 F 3	$*2\$$	r4 : $T \to F$
$ 0 E 1 + 6 T	$*2\$$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	$*2\$$	d7
$ 0 E 1 + 6 T 9 * 7	$2\$$	d5
$ 0 E 1 + 6 T 9 * 7 2 5	$\$$	r6 : $F \to$ nb
$ 0 E 1 + 6 T 9 * 7 F	$\$$	en 7 avec F : je vais en 10
$ 0 E 1 + 6 T 9 * 7 F 10	$\$$	r3 : $T \to T * F$
$ 0 E 1 + 6 T	$\$$	en 6 avec T : je vais en 9
$ 0 E 1 + 6 T 9	$\$$	r1 : $E \to E + T$
$ 0 E	$\$$	en 0 avec E : je vais en 1
$ 0 E 1	$\$$	ACCEPTÉ!!!

FIG. 6.5 – Analyse LR du mot $m = 3 + 4 * 2\$$

Remarques

Cette méthode permet d'analyser plus de grammaires que la méthode descendante (car il y a plus de grammaires SLR que LL)

•Dans cette méthode, ça n'a strictement aucune importance que la grammaire soit récursive à gauche, même au contraire, on **préfère**.
•Les grammaires ambiguës provoquent des **conflits**

- ✓ conflit **décalage/réduction** : on ne peut pas décider à la lecture du terminal *a* s'il faut réduire une production **S→α** ou décaler le terminal
- ✓ conflit **réduction/réduction** : on ne peut pas décider à la lecture du terminal *a* s'il faut réduire une production **S→α** ou une production T→**β**

On doit résoudre les conflits en donnant des **priorités** aux actions (d ou r) et aux productions.

Exemple

Soit la grammaire **E →E+E|E*E|(E)|nb**
Soit à analyser 3+4+5. Lorsqu'on lit le 2ème + on a le choix entre
Réduire ce qu'on a déjà lu par **E →E+E**. Ce qui nous donnera finalement le calcul (3+4)+5
Décaler ce +, ce qui nous donnera finalement le calcul 3+(4+5).
Ici c'est pareil. Mais bon, + est associatif à gauche, donc on préfèrera réduire. Soit à analyser 3+4*5. Lorsqu'on lit le * on a encore un choix shift/reduce. Si l'on réduit on calcule (3+4)*5, si on décale on calcule 3+(4*5) ! Il **faut** décaler.
Soit à analyser 3*4+5. On ne s'en fout pas non plus, il faut réduire !
Donc, il faut mettre quelque part dans l'analyseur le fait que * est prioritaire sur +.

Erreurs syntaxiques

Les erreurs syntaxiques révélées lorsque les unités lexicales provenant de l'analyseur lexical contredisent les règles grammaticales.
La nature de l'erreur est très difficile à déduire. La plupart du temps, le gestionnaire d'erreurs doit **deviner** ce que le programmeur avait en tête.
Il existe plusieurs stratégies de récupération sur erreur : **mode panique**, **au niveau du syntagme**, **productions d'erreur**, **correction globale**.
Une récupération inadéquate peut provoquer des erreurs qui n'ont pas été faites par le programmeur mais sont la conséquence du changement d'état de l'analyseur lors de la récupération sur erreur.

Récupération en mode panique

Quand il découvre une erreur, l'analyseur syntaxique élimine les symboles d'entrée les uns après les autres jusqu'à en rencontrer un qui appartienne à un ensemble d'unités lexicales de synchronisation, par exemple les délimiteurs (; , end ou }), dont le rôle dans un programme source est clair.
Bien que cette méthode saute en général une partie considérable du texte source sans en vérifier la validité, elle a l'avantage de la simplicité et ne peut pas entrer dans une boucle infinie.

Récupération au niveau du syntagme

Quand une erreur est découverte, l'analyseur syntaxique peut effectuer des corrections locales. Par exemple, remplacer une `,` par un `;`, un `wihle` par un `while`, insérer un `;` ou une `(`, ...Le choix de la modification à faire n'est pas évident en général. En outre, il faut faire attention à ne pas faire de modifications qui entraînerait une boucle infinie (par exemple décider d'insérer systèmatiquement un symbole juste avant le symbole courant).
L'inconvénient majeur de cette méthode est qu'il est pratiquement impossible de gérer les situations dans lesquelles l'erreur réelle s'est produite bien avant le point de détection.
On implante cette récupération sur erreur en remplissant les cases vides des tables d'analyse par des pointeurs vers des routines d'erreur. Ces routines remplacent, insèrent ou suppriment des symboles d'entrée et émettent les messages appropriés.

Productions d'erreur

Si l'on a une idée assez précise des erreurs courantes qui peuvent être rencontrées, il est possible d'augmenter la grammaire du langage avec des productions qui engendrent les constructions erronées.
Par exemple (pour un compilateur C) :
I→if E I (erreur : il manque les parenthèses)
I→if (E) then *I* (erreur : il n'y a pas de then en C)

Correction globale

Dans l'idéal, il est souhaitable que le compilateur effectue aussi peu de changements que possible. Il existe des algorithmes qui permettent de choisir une séquence minimale de changements correspondant globalement au coût de correction le plus faible. Malheureusement, ces méthodes sont trop coûteuses en temps et en espace pour être implantées en pratique et ont donc uniquement un intérêt théorique. En outre, le programme correct le plus proche n'est pas forcément celui que le programmeur avait en tête ...