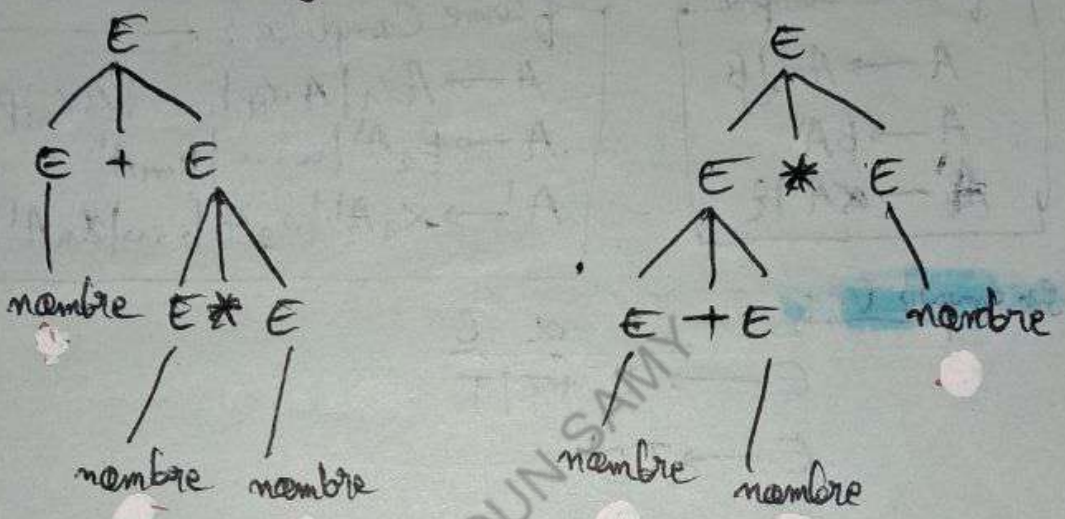
* Montrer que E est Ambigue:

- Il ya Ambiguité s'il existe au moins une ~~mot~~ phrase qui peut etre generée par plusieurs arbres de derivation ⟹ grammaire ambiguë.

Exemple 01: [illegible]

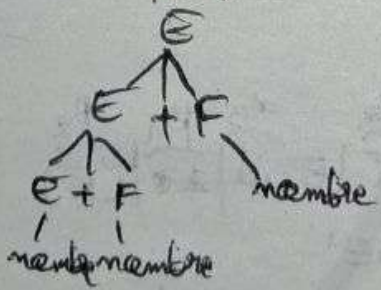


* Solution pour cette Ambiguité:

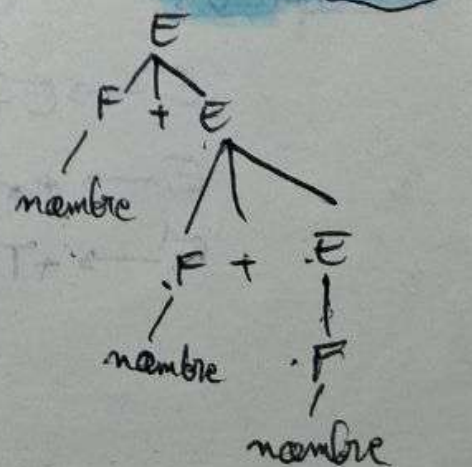
- choisir un coté pour la recursivité (gauche ou droit) et ajouter un non-terminal.

$E \rightarrow E+E$
$E \rightarrow nombre \mid (E)$ ⟹

à gauche {



$E \rightarrow E+F \mid F$
$F \rightarrow nombre \mid (E)$ {gauche



à droite:

$E \rightarrow F+E \mid F$
$F \rightarrow nombre \mid (E)$

* Elimination de Recursivité : (Gauche)

$$A \longrightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \beta_2 | \dots | \beta_m$$

- α et β des chaines quelconque (une suite de Terminaux et de non Terminaux).
- avec les β_m ne Commençant pas par A.

forme simple :

$$A \longrightarrow A\alpha | \beta$$
$$A \longrightarrow \beta A'$$
$$A' \longrightarrow \alpha A' | \varepsilon$$

forme Complexe :

$$A \longrightarrow A\alpha_1 | A\alpha_2 | \dots | A\alpha_n | \beta_1 | \dots | \beta_m$$
$$A \longrightarrow \beta_1 A' | \dots | \beta_m A'$$
$$A' \longrightarrow \alpha_1 A' | \alpha_2 A' | \dots | \alpha_n A' | \varepsilon$$

exemple 01 :

$$E \longrightarrow \overset{\alpha}{E+T} | \overset{\beta}{T}$$
$$E \longrightarrow TE'$$
$$E' \longrightarrow +TE' | \varepsilon$$

exemple 02 :

$$F \longrightarrow \overset{\alpha}{F id}$$
$$F \longrightarrow F'$$
$$F' \longrightarrow id F' | \varepsilon$$

exemple 03 :

$$E \longrightarrow \overset{\alpha_1}{E+T} | \overset{\alpha_2}{E*T} | \overset{\alpha_3}{E id} | \overset{\beta_1}{**} | \overset{\beta_2}{(E)}$$
$$E \longrightarrow **E' | (E)E'$$
$$E' \longrightarrow +TE' | *TE' | id E' | \varepsilon$$

* Factorisation: (Gauche)

- Si plusieurs productions commencent par le meme prefixe, lors de l'analyse il n'est pas evident de choisir la bonne production.

$$A \longrightarrow \alpha\beta_1 | \alpha\beta_2 | \dots | \alpha\beta_n | \gamma_1 | \gamma_2 | \dots | \gamma_n$$

$$A \longrightarrow \alpha A' | \gamma_1 | \gamma_2 | \dots | \gamma_n$$

$$A' \longrightarrow \beta_1 | \beta_2 | \dots | \beta_n$$

exemple 01:

$$E \longrightarrow \overbrace{+E}^{\alpha}\overbrace{dT}^{\beta_1} | \overbrace{+E}^{\alpha}\overbrace{*F}^{\beta_2} | \overbrace{+E}^{\alpha}\overbrace{id}^{\beta_3} | \overbrace{**}^{\gamma_1} | \overbrace{ab}^{\gamma_2}$$

$$E \longrightarrow +EA' | ** | ab$$

$$A' \longrightarrow dT | *F | id$$

* L'Analyse LL1:

- Deux Concepts se revelent necessaires pour faciliter l'analyse LL(1).

* L'ensemble Premier (First):

- on prends le premier symbole Terminal dans la production.

exemple 01:

$$\begin{cases} S \longrightarrow iEtSS' | a \\ S' \longrightarrow eS | \varepsilon \\ E \longrightarrow b \end{cases}$$

first(S) = i, a
first(S') = e, ε
first(E) = b

exemple 02:

$$\begin{cases} E \to TE' \\ E' \to +TE' / -TE' / \varepsilon \\ T \to FT' \\ T' \to *FT' / /FT' / \varepsilon \\ F \to nb / (E) \end{cases}$$

first (E) = nb, (
first (E') = +, -, ε
first (T) = nb, (

first (T') = *, /, ε
first (F) = nb, (

* L'ensemble Suivant (Fallow):

Fallow (X)

- X Terminal → Fallow(X) = le symbol terminal
- X (gride) → Fallow (X) = Fallow (origine de braduction)
- X non Terminal → Fallow (X) = Eprst (non Terminal)
  - Si ε ∈ Eprst (non Terminal) + Fallow (origine de Braduction)

Exemple 01:

$$\begin{cases} S \to iEtSS' / a \\ S' \to eS / \varepsilon \\ E \to b \end{cases}$$

Fallow (S) = $e
Fallow (S') = $e
Fallow (E) = t

exemple 02:

$$\begin{cases} E \to TE' \\ E' \to +TE' \mid -TE' \mid \varepsilon \\ T \to FT' \\ T' \to *FT' \mid /FT' \mid \varepsilon \\ F \to nb \mid (E) \end{cases}$$

follow(E) = {), $}
follow(E') = {), $}
follow(T) = {+, -,), $}
follow(T') = {+,), $}
follow(F) = {*, /, +, -,), $}

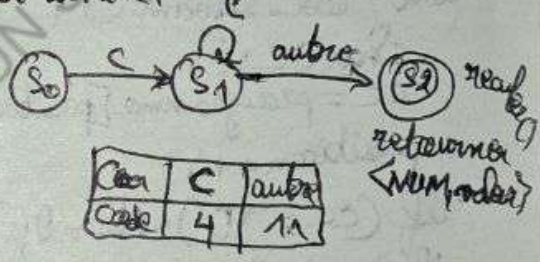
Remarque : Une grammaire est LL1 à gauche quand elle n'est pas Ambiguë, n'est pas recursive et elle est factorisée à gauche.

* Les fragements de code:

- quand on a une boucle on fait while:

```
case 1: CC = car_suivant();
        while (CC == 4)
            CC = car_suivant()
        etat = 2;
        break;
```

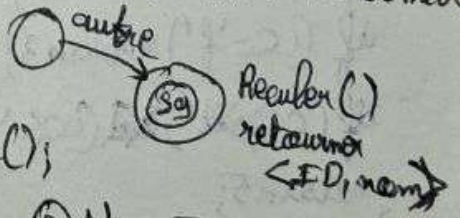
```
if (CC=4)
    etat=1
else
    etat=2
```



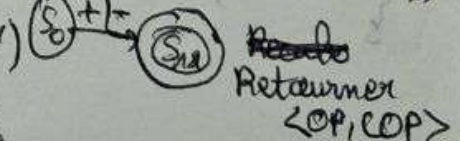
Car	c	autre
Code	4	11

- la fonction get_lexeme() permettant de retourner le dernier lexeme reconnu (sous chaine):

```
case 9: reculer();
        attribut.nom = get_lexeme();
        return ID;
```




```
Case 12: if(get_lexeme()[0] == '+')
             attribut.cop = PLUS;
         else attribut.cop = MOINS;
         return OP;
```



- Pour une valeur : attribut.valeur = valeof(get-Perme()) (6)

attribut.valeur = atof(get-Perme())

valeof : Convertir les chaînes ~~constantes~~ entières en Valeurs.

atof : Convertir les chaîne de chiffre hexadecimaux en Valeurs.

```
Case 7 : reculer();
        attribut.valeur = atoi(get-Perme());
        return NUM;
```

(S7) Reculer(); Retourner <NUM, valeur>

* Autre Cas:

~~Case 6 return~~

```
Case 6 : return PV;
```

(S6) retourner <PV, >

* fonction Car_suivant();

```
int car_suivant() {
   char cc;
   cc = programme[position]
   position++;
   if (cc='\0') return 0;
   if (cc='$') return 1;
   if (cc='i') return 2;
   if (cc='f') return 3;
   if (cc>='a') && (cc<='z') return 4;
   return 5;
}
```

* fonction Token_suivant();

```
enum token_suivant(){
  int cc;
  int etat;
   etat = ...;
   switch (etat){
   case 0: cc=car_suivant();
     switch (cc){
       case 0: etat=5; break;
       ⋮
       case 4: etat=4; break;
       default: etat=12;
     }
   break
  case 1: cc = car_suivant();
       if (cc==1) etat=0;
       else etat=1;
    break;
  case 2: cc=car_suivant()
     if(cc==2)||(cc==3) etat=4
     else etat=6
  case 5: return FIN;
  case 7: reculer() return IF;
```

Plus de 2 instructions

* Question...

*Role Analy...

caractere ...

en une sequ...

la Table de ...

*Role Anal...

retournée p...

langage Con...

~~*Phase ...~~

* Phase d'...

Programme ...

→ Gestion du... intermé...

* Auto Con...

tion est le ...

est differen...

* Role édite...

programme ...

* Difference...

de la form...

intermedia...

son execu...

*Role parti...

puis Const...

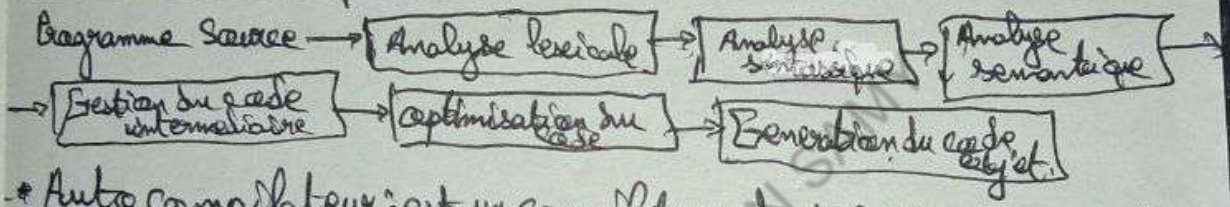
* Questions de Cours:

*Role Analyse lexicale: lire le programme source caractere par caractere, sauter les blancs et les commentaires, grouper les caracteres en une sequence d'unités lexicales (les symboles du langage), construire la Table de symbole.

*Role Analyse syntaxique: consiste à verifier que la sequence retournée par la phase precedente est conforme à la syntaxe du langage considéré.

~~*Phase d'un Compilateur:~~

*Phase d'un Compilateur:



*Auto compilateur: est un compilateur dont le langage d'implementation est le même que le langage source et dont la machine cible est differente de celle sur laquelle on le developpe.

*Role Editeur de liens: sert à resoudre les references externes d'un programme.

Difference entre un Compilateur et interpreteur à propos du Traitement de la forme intermediaire: un Compilateur ~~~~ traduit la forme intermediaire en programme cible alors qu'un interpreteur effectue son execution et ne genere aucun code cible.

*Role partie synthese d'un Compilateur: optimise le code intermediaire puis construit le programme cible pour la machine choisie.

(3)

* Quels prétraitements fait subir un preprocesseur à un prgm source ?

- la gestion des fichiers inclus.
- le développement des macro-instructions.
- le traitement de la Compilation Conditionnelle.

* Inconvenient de l'édition des liens dynamiques : est que le prgm exécutable résultant est dépendant d'un ensemble de bibliothèques partagées sans lesquelles il ne peut pas être exécuté.