الكلية متعددة التخصصات - ورزازات
ⵜⴰⵖⵣⵍⵓⵜ ⵜⴰⵅⵜⵣⴰⵎⵉⵣⵜ – ⵍⵓⵕⵣⵓⵕⵜ
FACULTÉ POLYDISCIPLINAIRE DE OUARZAZATE
UNIVERSITÉ IBN ZOHR - AGADIR

# Compilation

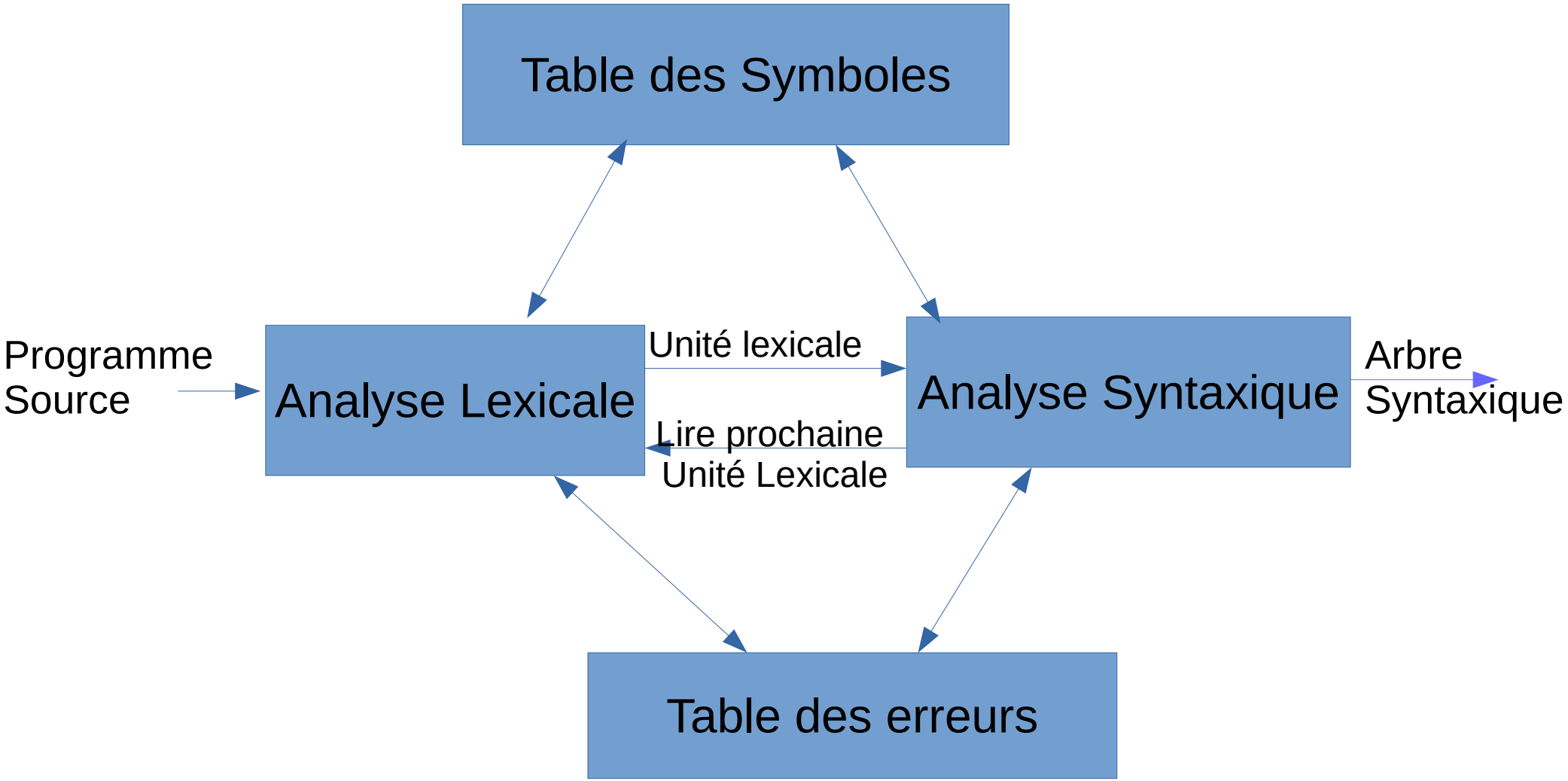
## Chapitre3 : Analyse syntaxique

SMI/S5
Compilation
Prof. : M. BENADDY

# Introduction

- L'analyse syntaxique (A.S.) est l'une des opérations majeures d'un compilateur, qui consiste à indiquer si un texte est grammaticalement correcte
- et à en tirer une représentation interne qu'on appelle arbre syntaxique (arbre d'analyse),
- l'A.S. obtient une chaîne d'unité lexicale de l'analyse lexicale et vérifie que la chaîne peut être engendrée par la syntaxe du langage source, celle-ci est spécifiée par une grammaire.

# Introduction



# Introduction

- Il existe trois types généraux d'analyseurs syntaxiques :
  - Méthodes universelles
    - Descendantes (construction de l'arbre syntaxique en partant de la racine vers les feuilles)
    - Et ascendantes (construction de l'arbre syntaxique en partant des feuilles vers la racine).
  - Les méthodes d'analyse universelles tel l'algorithme de Cocke-Younger-Kasami et l'algorithme d'Earley, ces méthodes sont peu utilisées dans la production de compilateurs, pour leur inefficacité.

# Grammaire hors contexte

- Une grammaire G est donnée sous forme d'un quadruplet <V, ∑,R,S> où :
  - V est un vocabulaire appelé aussi alphabet, c-à-d un ensemble fini de symboles.
  - $\sum \subseteq V$ Est l'ensemble des symboles terminaux, c-à-d les symboles de base du langage à partir desquels les chaînes sont formées. L'expression d'unité lexicale est synonyme de terminal quand on parle de grammaire pour les langages de programmation.
  - V-∑ est l'ensemble des symboles non terminaux,  c-à-d les variables syntaxiques qui dénotent un ensemble de chaînes qui aident à spécifier le langage engendré par la grammaire.
  - R est l'ensemble des règles de production de la grammaire, qui spécifie la manière dont les terminaux et les non-terminaux qui peuvent être combinés pour former des chaînes. Chaque règle de production consiste en un non terminal suivi d'une flèche suivie d'une chaîne formée de terminaux et non-terminaux.
  - $S \subset \sum - V$ Est l'axiome de la grammaire, c-à-d un symbole non terminal particulier tel que l'ensemble des chaînes qu'il dénote est le langage défini par la grammaire.

# Grammaire hors contexte

- **Exemple :** Soit la grammaire constituée des productions suivantes :

  exp → exp op exp
  exp → (exp)
  exp → -exp
  exp → id
  op → +
  op → -
  op → *
  op → /

Les symboles terminaux sont +, -, *, /, id, ()

Les symboles non-terminaux sont exp, op, …

# Grammaire hors contexte

- Notation :
  - Les symboles suivants sont des terminaux :
    - Les lettres minuscules du début de l'alphabet a, b, c, …
    - Les opérateurs +, -, *, /
    - Les symboles de ponctuation (() , ; …)
    - Les chiffres 0, à 9
    - Les lettres minuscules représentants les chaînes terminales x, u, v, w, …
  - Les symboles suivants sont des non-terminaux :
    - Les lettres majuscules de l'alphabet A, B, C,
    - La lettre S désigne un axiome
    - Les lettres majuscules de la fin de l'alphabet X, Y, Z représentent les symboles terminaux et non terminaux.
    - Les lettres grecque α, β, δ, … représentent les chaînes de symboles grammaticaux. Une production générique peut être écrite de la façon suivante : A → α, où A est la tête et α est le corps.
    - Le mot vide s'exprime par ε

      E → E A E | (E) | -E | a

      A → + | - | * | /

# Grammaire hors contexte

- **Dérivation :**
- L'idée de dérivation est de traiter une règle de production comme une règle de réécriture dans laquelle le non-terminal en partie gauche est remplacé par la chaîne en partie droite de la règle de production.
- **Exemple**, soit la grammaire suivante : E → E + E | E * E | (E)| -E | a
  - la règle E → -E signifie qu'une expression précédée d'un signe – est aussi une expression, cette expression peut engendrer des expressions plus complexes en remplaçant l'occurrence de E par -E. Cette action peut être décrite par E => -E et se lit E se dérive en -E
  - La règle E → (E) nous permet de remplacer E par (E) dans une chaîne quelconque de symboles grammaticaux
  - E * E => (E) * E, E * E => E * (E)

# Grammaire hors contexte

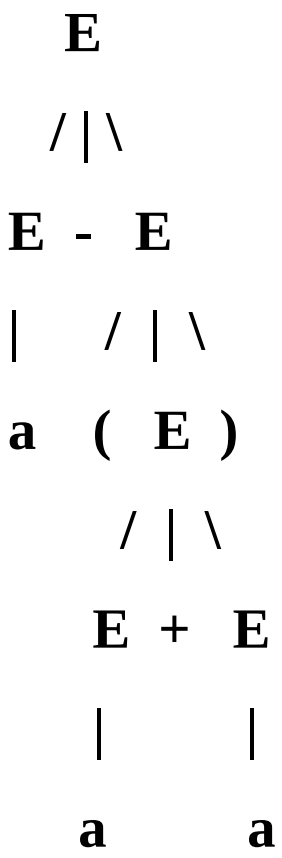
- **Dérivation :**
- On peut prendre un E unique et appliquer répétitivement les règles de production dans un ordre quelconque est obtenir une séquence de remplacement.
- E => (E) => -(E) => -(a)
- Si $\alpha_1 => \alpha_2 => \alpha_3 => ... => \alpha_n$ on dit que $\alpha_1$ se dérive en $\alpha_n$
- Le symbole => signifie se dérive en une seule étape.
- Le symbole =*> signifie se dérive en 0, une ou plusieurs étapes, alors on peut écrire E =*> -(a)
- De même le symbole =+> signifie se dérive en une ou plusieurs étapes.

# Grammaire hors contexte

- **Dérivation : Exemples :** Soit la grammaire suivante définit dans l'exemple précédent :
  - La chaîne -(a+a) est une phrase de la grammaire car on a les dérivations suivantes : E => -E => -(E) => -(E+E) => -(a+E) => -(a+a) et note alors E =*>-(a+a)
  - E =*> (cte + a) est fausse car cte ne fait pas partie de l'alphabet.
  - E =*> -(a - a) est fausse car (E – E) ne fait pas partie de l'alphabet,
  - On a aussi E => -E => -(E) => -(E+E) => -(E + -E) =*> -(a + -a)
  - À chaque étape de la dérivation on doit faire 2 choix. Il faut choisir le non-terminal à remplacer, par exemple dans la grammaire de l'exemple précédent, la dérivation peut se poursuivre comme suite : E => E + E => E + a => a + a
  - On constate qu'on a changé le non-terminal droite. Si on considère les dérivations dans lesquelles seul le non-terminal gauche est remplacé, de telles dérivations sont appelées dérivations gauches et on peut écrire : E =*>$_g$ (a + a)
  - la définition reste valide pour les dérivations droites et on note E =*>$_d$ (a + a)

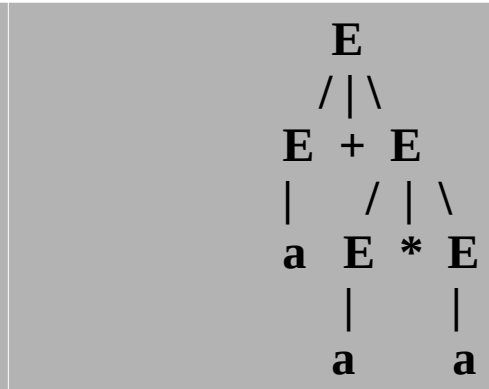
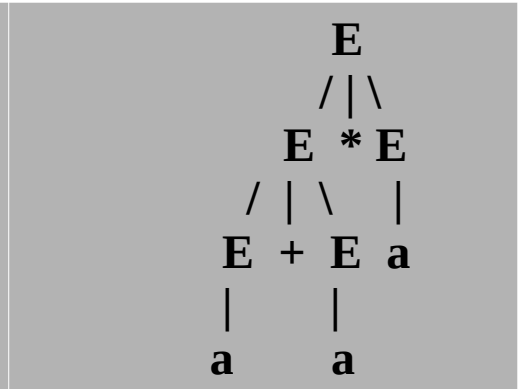
# Grammaire hors contexte

- **L'arbre syntaxique :** Un arbre syntaxique ou un arbre d'analyse est une représentation graphique des dérivations successives aboutissants à une expression du langage, par exemple l'arbre syntaxique correspondant à la phrase a – (a + a) est donné par le graphe suivant :



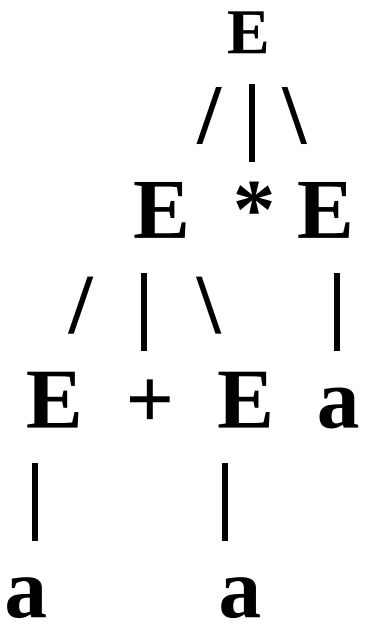
# Grammaire hors contexte

- **Ambiguïté :** Une grammaire est dite ambiguë s'il existe plusieurs arbres syntaxiques de dérivation distincts, autrement dit c'est une grammaire qui produit plus d'une dérivation gauche ou bien droite pour la même phrase.
- Pour certains types d'analyseurs il est nécessaire de rendre la grammaire non ambiguë car on ne peut pas déterminer de façon unique l'arbre syntaxique associé à une phrase dérivant d'une grammaire ambiguë. Soit la phrase a + a*a

| | | | |
|---|---|---|---|
| **E =*> a + a * a**<br>**E => E + E**<br>**E => E + E * E**<br>**E => a + E * E**<br>**E => a + a * E**<br>**E => a + a * a** |  | **E =*> a + a * a**<br>**E => E * E**<br>**E => E+ E * E**<br>**E => a + E * E**<br>**E => a + a * E**<br>**E => a + a * a** |  |

# Grammaire hors contexte

- **Ambiguïté :** Si on tient compte du faite que l'opérateur * est prioritaire que l'opérateur + alors l'ambiguïté est levée on optera pour l'arbre syntaxique suivante :



# Grammaire hors contexte

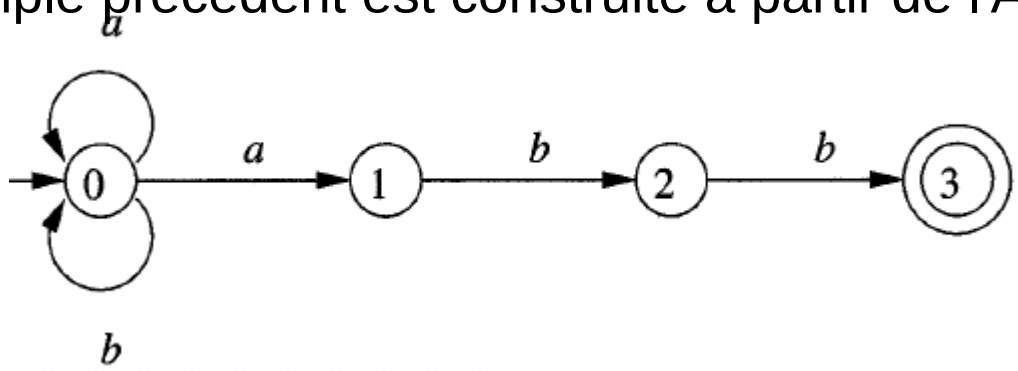
- **Grammaire hors contexte versus expressions régulières :**
  - La grammaire est une notation efficace que les expressions régulières.
  - Une construction qui peut être décrite par une expression régulière peut être décrite par une grammaire mais pas le contraire.
  - Alternativement chaque langage régulier, est un langage hors contexte, le contraire étant incorrecte
  - **Exemple** : l'expression régulière (alb)*abb et la grammaire

$$
\begin{aligned}
A_0 &\rightarrow aA_0 \mid bA_0 \mid aA_1 \\
A_1 &\rightarrow bA_2 \\
A_2 &\rightarrow bA_3 \\
A_3 &\rightarrow \epsilon
\end{aligned}
$$

  - Décrivent le même langage, qui est l'ensemble des mots formés de a et b qui se terminent par abb

# Écriture d'une grammaire

- **Expressions régulières vers grammaire hors contexte :**
  - Une grammaire qui reconnaît un langage comme un AFN peut être construite à partir celle-ci.
  - La grammaire de l'exemple précédent est construite à partir de l'AFN suivant




  - Pour chaque état i de l'AFN créer un non terminale $A_i$
  - Si un état i a une transition vers un état j sur l'entrée a, ajouter la production $A_i \rightarrow aA_j$.
  - Si l'état i par vers l'état j sur ε, ajouter la production $A_i \rightarrow A_j$
  - Si i est un état d'acceptation ajouter $A_i \rightarrow \varepsilon$
  - Si $A_i$ est l'état de départ de l'AFN, alors $A_i$ est l'axiome de la grammaire.

# Écriture d'une grammaire

- **Élimination de la récursivité à gauche :**
  - Une grammaire est récursive à gauche si elle contient au moins un non terminal A telle qu'il existe une dérivation.

    A => Aα  => βα (α chaîne)

    $A =>^{+} A\alpha$
  - Les méthodes d'analyse descendante ne peuvent pas fonctionner avec une grammaire récursive à gauche on a donc besoin d'une transformation grammaticale qui élimine cette récursivité à gauche.
  - On peut éliminer une production récursive en la réécrivant.

# Écriture d'une grammaire

- **Élimination de la récursivité à gauche :**
- Considérons le non-terminal A dans les deux productions
  - $A \rightarrow A\alpha|\beta$ où α et β sont deux suites de terminaux et de non-terminaux qui ne commencent pas par A.
- On peut obtenir les mêmes productions en réécrivant les productions définissant A de la manière suivante

  $A \rightarrow \beta A'$

  $A' \rightarrow \alpha A'|\varepsilon$

  où A' est un nouveau non-terminal.
- Cette réécriture n'altère pas l'ensemble des chaînes dérivables depuis A.

# Écriture d'une grammaire

- **Élimination de la récursivité à gauche :**
- **Exemple** : Éliminer la récursivité à gauche de la grammaire suivante :

  E → E+T|T

  T → T*F|F

  F → (E)|id

# Écriture d'une grammaire

- **Élimination de la récursivité à gauche :**
- **Exemple** : Éliminer la récursivité à gauche de la grammaire suivante :

$E \rightarrow E+T|T$

$T \rightarrow T*F|F$

$F \rightarrow (E)|id$

**Solution►**

$E \rightarrow TE'$

$E' \rightarrow +TE'|\varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT'|\varepsilon$

$F \rightarrow (E) \mid id$

# Écriture d'une grammaire

- **Élimination de la récursivité à gauche :**
- D’une manière générale quelque soit le nombre de production de A :

$$A \rightarrow A\alpha_1 | A\alpha_2 | \ldots | A\alpha_n | \beta_1 | \beta_2 | \ldots | \beta_m$$

$$A \rightarrow \beta_1 A' | \beta_2 A' | \ldots | \beta_m A'$$

$$A' \rightarrow \alpha_1 A' | \alpha_2 A' | \ldots | \alpha_n A' | \varepsilon$$

# Écriture d'une grammaire

- **Factorisation à gauche :** La factorisation à gauche est une transformation grammaticale utile pour obtenir une grammaire convenant à l'analyse prédictive.
- En générale si on a la production
- $A \rightarrow \alpha\beta_1 | \alpha\beta_2$ et si l'entrée commence par une chaîne non vide dérivée de α, on ne saura pas si il faut développer A en $\alpha\beta_1$ ou $\alpha\beta_2$ cependant il est possible de faire une factorisation à gauche telle que $A \rightarrow \alpha A'$, $A' \rightarrow \beta_1 | \beta_2$
- D'une façon générale si on a $A \rightarrow \alpha\beta_1 | \ldots | \alpha\beta_n | \delta$ où δ représente toute chaînes qui ne commence pas par α alors on aura
  - $A \rightarrow \alpha A' | \delta$ A' non-terminal
  - $A' \rightarrow \beta_1 | \ldots | \beta_n$

# Écriture d'une grammaire

- **Factorisation à gauche :**
- **Exemple :** Factoriser :

$$S \rightarrow iEtS \mid iEtSeS \mid a$$

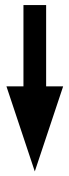
$$E \rightarrow b$$

# Écriture d'une grammaire

- **Factorisation à gauche :**
- **Exemple :** Factoriser :

$S \rightarrow iEtS \mid iEtSeS \mid a$

$E \rightarrow b$



$S \rightarrow iEtSS' \mid a$

$S' \rightarrow eS \mid \varepsilon$

$E \rightarrow b$

# Processus d'analyse

- **L’analyse descendante :** dans laquelle on construit l’arbre en descendant de la racine vers les feuilles, c-à-d on essaie à partir de l’axiome de la grammaire de dériver le programme source, il s’agit de dériver à chaque étape qu’il est la règle qui sert à engendrer le mot qu’on lit.
- **L’analyse ascendante** : dans laquelle on construit l’arbre syntaxique en montant des feuilles vers la racine, elle correspond au parcours inverse du parcours descendant, c-à-d on établit des réductions sur la chaîne à analyser pour aboutir à l’axiome de la grammaire. Cette méthode est puissante mais plus complexe à mettre en œuvre que l’analyse descendante.

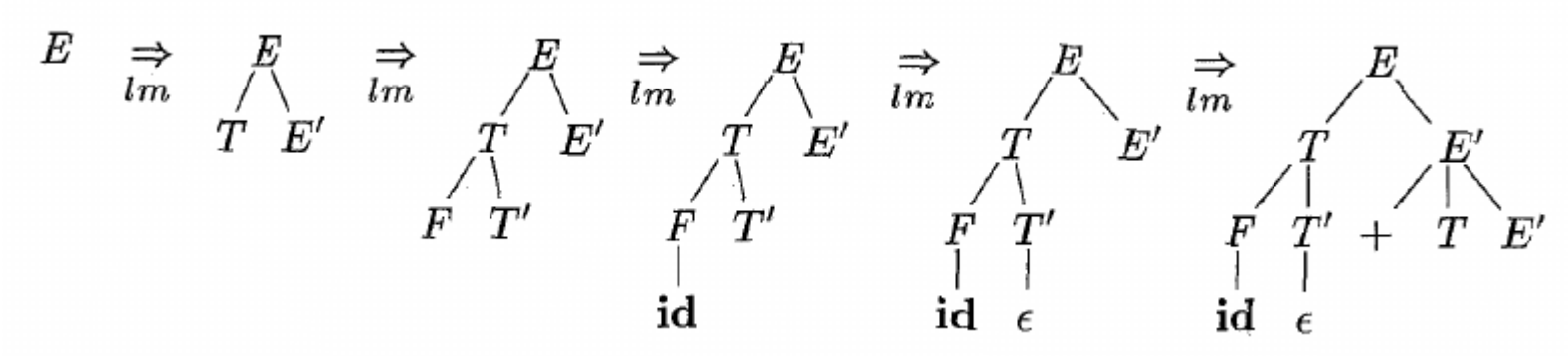
# L’analyse descendante

- **L'analyse descendante :** On effectue la construction descendante d’un arbre syntaxique en partant de la racine, étiqueté par l’axiome, et en relisant de manière répétitive les deux étapes suivantes :
  - Au nœud n étiqueté par le non-terminal A, choisir une des règles de production définissant A et construire les fils de n avec les symboles en partie droite de la production.
  - Déterminer le prochain nœud où un sous-arbre doit être construit.

# L’analyse descendante

- **L'analyse descendante :**
- **Exemple :** l'arbre syntaxique de la chaîne id + id * id

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' | \varepsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' | \varepsilon$$
$$F \rightarrow (E) \mid id$$



# L'analyse descendante

- **L'analyse descendante :**
- **Exemple :** l'arbre syntaxique de la chaîne id + id * id

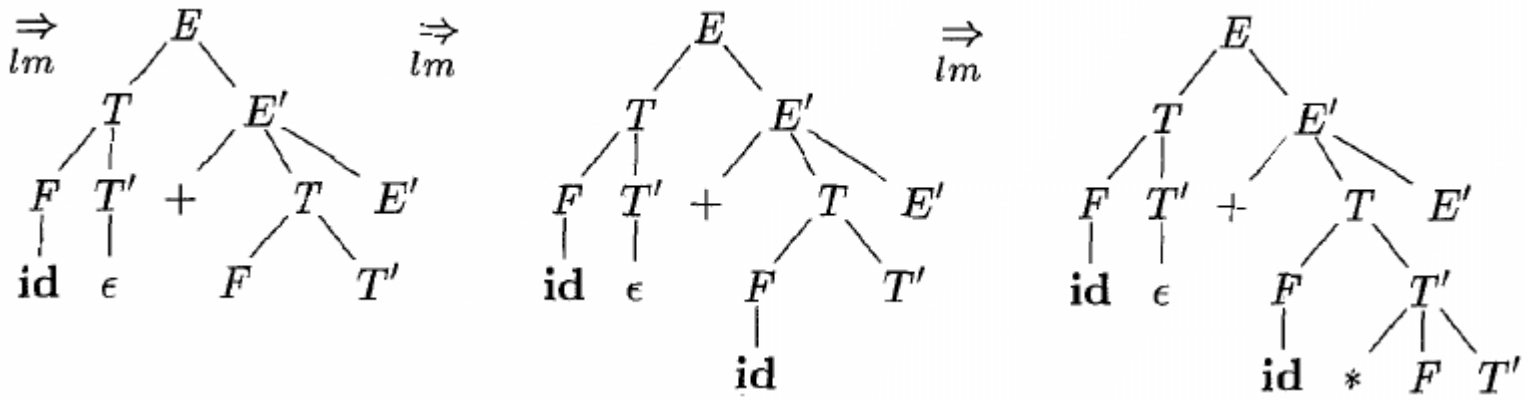
$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) \mid id$



# L’analyse descendante

- **L'analyse descendante :**
- **Exemple :** l'arbre syntaxique de la chaîne id + id * id

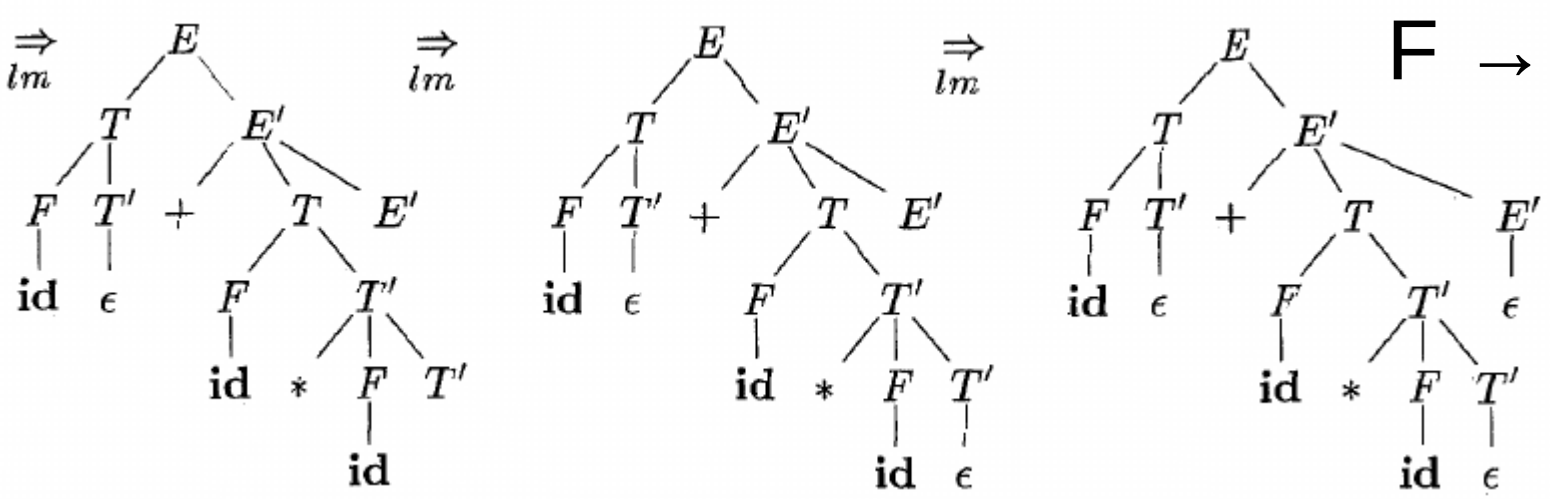
$E \rightarrow TE'$

$E' \rightarrow +TE' | \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' | \varepsilon$

$F \rightarrow (E) \mid id$

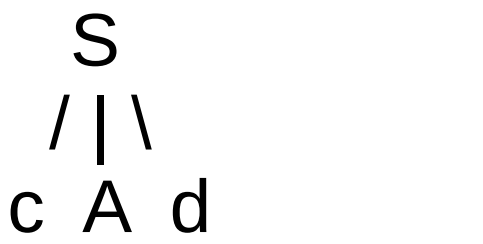


# L’analyse descendante

- **L'analyse descendante récursive :** L’analyse syntaxique par descente récursive est une méthode d’analyse syntaxique descendante dans laquelle on exécute un ensemble de procédure récursive pour traiter l’entrée en question. Cette analyse peut impliquer des retours arrière.
- **Exemple** : Soit la grammaire définit par S → cAd , A → ab|a et soit la chaîne W = cad

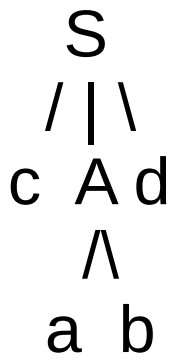
# L’analyse descendante

- **L'analyse descendante récursive :**
- **Exemple** : Soit la grammaire définit par S → cAd , A → ab|a et soit la chaîne W = cad
- Pour construire de façon descendante un arbre d’analyse syntaxique pour la chaîne W on construit initialement un arbre qui contient un seul nœud S.
- Un pointeur d’entrée repère c, le premier symbole de W. on utilise alors la première règle de production de S, pour aboutir à l’arbre suivant :



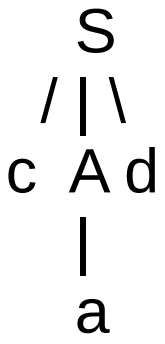
# L’analyse descendante

- **L'analyse descendante récursive :**
- **Exemple** : Soit la grammaire définit par S → cAd , A → ab|a et soit la chaîne W = cad
- La feuille la plus à gauche étiquetée c correspond au premier symbole de W.
- Par conséquent on avance le pointeur d’entrée sur a, seconde symbole de W et on considère la feuille étiquetée a on peut développer A en utilisant la deuxième règle pour obtenir l’arbre suivant :



# L’analyse descendante

- **L'analyse descendante récursive :**
- **Exemple** : Soit la grammaire définit par S → cAd , A → ab|a et soit la chaîne W = cad
- On a une concordance avec le second symbole d’entrée a, et on compare d avec la feuille suivante étiquetée b comme b != d on signale un échec et on retourne à A pour voir s’il existe une autre alternative non encore essayer.
- En retournant à A on doit remettre le pointeur d’entrée en position 2. On essaie la seconde alternative de A et on obtient l’arbre suivant :



# L’analyse descendante

- Notion de premier et de suivant :
  - La construction d'un analyseur syntaxique descendant est facilitée, car deux fonctions associées à une grammaire G sont premier() et suivant().
  - Ces fonctions nous permettent si possible de remplir les entrées de la table d'analyse descendante pour la grammaire G.
- Premier : Si α est une chaîne de symboles grammaticaux premier de α désigne l'ensemble des terminaux qui commencent les chaînes qui se dérivent de α.
- Si X est un terminal alors premier(X)={X}.
- Si X est un non-terminal et que X → Y1Y2....Yk est une producrion pour k>=1, alors ajouter a à premier(X) si pour certain i, a Є premier(Yi), et ε est dans premier(Yi),...,premier(Yi-1) ; c-à-d que Y1Y2....Yi-1=*> ε. Si ε Є premier(Yj) pour j=1,2,…,k, alors ajouter ε à premier(X). la règle s'applique ainsi de suite sur tout les éléments de la production.
- Si α=*> ε alors ε est aussi dans l'ensemble premier(α).

# L’analyse descendante

- **Notion de premier et de suivant :**
- **Suivant** : Pour chaque symbole non-terminal A, suivant(A) définit l'ensemble des terminaux qui peuvent apparaître immédiatement à droite de A dans toute chaînes de symboles constituant les règles de production.
  - Si A est le symbole le plus à droite d'une chaîne alors le symbole $ est ajouté à l'ensemble suivant de A. pour déterminer suivant(A) où A est non-terminal il faut suivre les étapes suivante jusqu'à ce que aucun élément ne puisse être ajouté à suivant(A) :
  - si A est l'axiome de la grammaire alors ajouter $ à suivant de A.
  - si la règle X → AaB $\epsilon$ G si a est un terminal alors ajouter a à suivant(A).
  - si X → αA est une règle de la grammaire, alors ajouter l'ensemble suivant(X) à suivant(A).
  - si A → αBβ est une règle de la grammaire le contenu de premier(β) à l'exception de ε est ajouté à suivant(B).
  - si il existe une règle de la grammaire A → αBβ tel que premier(β) contient ε, les éléments de suivant(A) sont ajoutés à suivant(B).

# L’analyse descendante

- **Notion de premier et de suivant :**
- **Exemple** : Soit la grammaire suivante :

  E → TE'

  E' → +TE' | ε

  T → FT'

  T' → *FT' | ε

  F → (E)|id

  - Déterminer premier(E), premier(E'), premier(T'), suivant(E), suivant(E'), premier(T), suivant(T), suivant(T'), suivant(F), premier(F)

# L’analyse descendante

- **Notion de premier et de suivant :**
- **Exemple** : Soit la grammaire suivante :
  - E → TE' , E' → +TE' | ε , T → FT' , T' → *FT' | ε , F → (E)|id
  - premier(E)=premier(T)=premier(F)={(,id}, premier(E')={+,ε}, premier(T')={*,ε}, suivant(E)={),$}, suivant(E')={), $}
  - Puisque E est un axiome alors $ ϵ suivant(E) puisque F → (E) de la ) ϵ suivant(E).
  - La règle E → TE' et la 3ème étape de la construction de suivant on a suivant(E) ϵ suivant(E') d'où suivant(E) = suivant(E') ={),$} suivant(T)={+,),$}=suivant(T')
  - Pour suivant(T) on a la règle E' → +TE' et E' => ε c-à-d que premier(E') à l'exception de ε ϵ suivant(T) suivant(F)={*,+,),$}

# Grammaire LL(1)

- Une grammaire dont la table d'analyse n'a aucune entrée définit de façon multiple est appelée une grammaire LL(1).
- Le premier L désigne le parcours de l'entrée de gauche à droite (Letf to right scanning), le second L signifie dérivation gauche (Left most derivation) le (1) indique qu'on utilise un seul symbole d'entrée à chaque étape nécessitant la prise d'une décision d'action d'analyse.

# Grammaire LL(1)

- Une grammaire G est dite LL(1) si ses règles de production vérifient les conditions suivantes :
- Si A → α et A → β sont deux règles de G alors on doit avoir soit $\alpha =^{*}> \varepsilon$ soit $\beta =^{*}> \varepsilon$ et non pas les deux
- Si A → α et A → β sont deux règles de G alors $premier(\beta) \cap premier(\alpha) = \emptyset$
- Si A → α et A → β sont deux règles de G et si ε est dans premier(β) ou dans premier(α) alors $suivant(A) \cap premier(\alpha) = \emptyset$
- Ces conditions entraînent l'unicité d'existence d'une règle dans une entrée de la table d'analyse prédictive.
- En plus de ces conditions une grammaire est LL(1) si toute ses règles de production sont non récursives à gauche et factorisables.

# Grammaire LL(1)

- **Algorithme de Construction de la table d'analyse prédictive :**
- **Données** : la grammaire G
- **Résultat** : table d'analyse prédictive M pour G
- **Méthode** :
- pour chaque règle de la production A → α de la grammaire procéder aux étapes :
  - pour chaque a dans premier(α) placer la règle A → α dans la position M(A,a)
  - si ε ϵ premier(α), placer A → α dans M(A,b) pour chaque terminal b qui ϵ suivant(A).
  - si ε ϵ premier(α) et $ ϵ suivant(A) placer la règle A → α dans M(A,$)
- faire de chaque entrée non définit de M dans la table une erreur.

# Grammaire LL(1)

- **Algorithme de Construction de la table d'analyse prédictive :**
- **Exemple :** soit la grammaire définit par les règles suivantes :

  E → TE' , E' → +TE' | ε , T → FT' , T' → *FT' | ε , F → (E)|id
- premier(TE')=premier(T)={id,(} c-à-d M(E,id)=E → TE', M(E,( ) = E → TE'
- E' → +TE' premier(+TE')={+} c-à-d M(E',+)= E' → +TE'
- E' → ε , suivant(E')={),$} donc M(E',) ) = E' → ε et M(E',$) = E' → ε
- T → FT', premier(FT') = premier(F) = {(,id} donc M(T,( )= T → FT' et M(T,id) = T → FT'
- T' → *FT', premier(*FT') = {*} donc M(T',*) = T' → *FT'
- T' → ε , suivant(T')={+, ) , $}donc M(T',+) = T' → ε, M(T',) )= T' → ε et m(T', $)=T' → ε
- F → (E) , premier((E))={(} alors M(F,( )= F → (E)
- F → id , premier(id)={id} alors M(F,id)= F → id

# Grammaire LL(1)

- **Algorithme de Construction de la table d'analyse prédictive :**
- **Exemple :** soit la grammaire définit par les règles suivantes :

  E → TE' , E' → +TE' | ε , T → FT' , T' → *FT' | ε , F → (E)|id

| Non-terminal | Symbole d'entrée | | | | | |
|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ |
| E | E → TE' | | | E → TE' | | |
| E' | | E' → +TE' | | | E' → ε | E' → ε |
| T | T → FT' | | | T → FT' | | |
| T' | | T' → ε | T' → *FT' | | T' → ε | T' → ε |
| F | F → id | | | F → (E) | | |



Analyse id+id*id

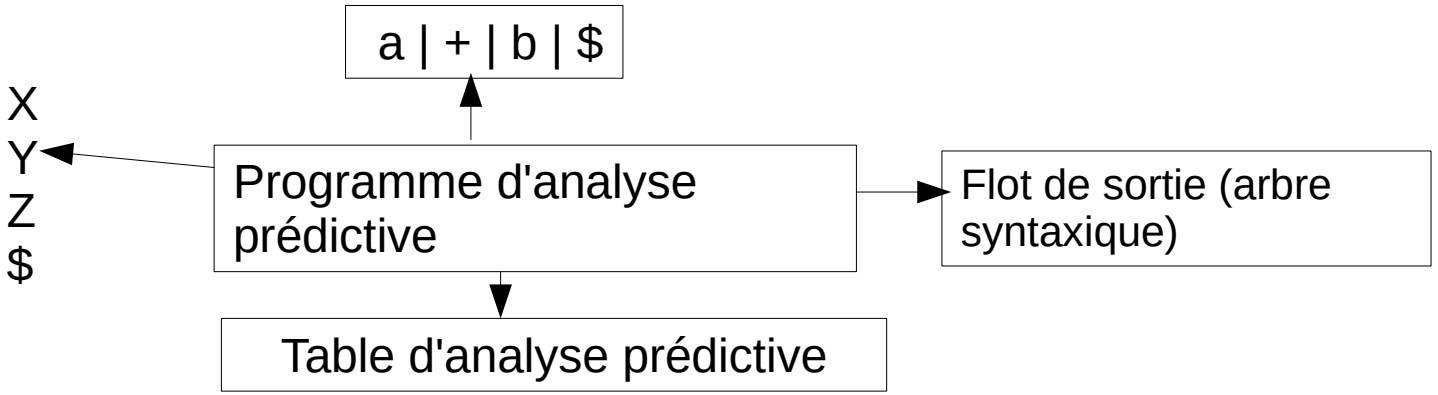
# Grammaire LL(1)

- **Algorithme de Construction de la table d'analyse prédictive :**
- **Exercices d'application :**Donner les premier, suivant et les tables prédictive ?

| 1. soit la grammaire suivante : | 2. soit la grammaire suivante |
|---|---|
| E → E+T\|T<br>T → T*F\|F<br>F → (E)\|id | S → iEtSS' \|a<br>S' → eS \| ε<br>E → b |

# Grammaire LL(1)

- **Analyse prédictive :**
- L'analyseur syntaxique prédictive possède un tampon d'entrée, une pile, une table d'analyse prédictive et un flot de sortie.
- Le tampon d'entrée contient la chaîne à analyser suivie du symbole $, qui est utilisé comme marqueur d'extrémité droite indiquant la fin de la chaîne d'entrée.
- La pile contient une séquence de symboles grammaticaux avec le symbole $ marquant la fin de la chaîne de la pile. La table d'analyse est une table à deux dimensions qu'on notera M(A,a) où A est un non-terminal et *a* est un terminal ou bien le symbole $.
- Les éléments de la table contiennent éventuellement des règles de la grammaire :



# Grammaire LL(1)

- **Algorithme d'analyse prédictive :**
- Avant de commencer l'analyse d'une chaîne de caractères, il faut convertir la grammaire décrivant le langage qui accepte la chaîne à analyser en une grammaire LL(1), ensuite construire la table prédictive de cette dernière. L'analyse syntaxique est contrôlée par un programme qui a le comportement suivant :
- On considère X le symbole en sommet de la pile et a le symbole courant de la chaîne à analyser. Ces deux symboles déterminent l'axiome de l'analyseur. Il existe trois possibilités :
- Si X = a = $, l'analyseur s'arrête et annonce le résultat final de l'analyse.
- Si X = a ≠ $, l'analyseur enlève X de la pile et avance son pointeur d'entrée sur le symbole suivant.
- Si X est non-terminal le programme d'analyse consulte l'entrée M(X,a) de la table d'analyse prédictive, cette entrée sera soit une X production de la grammaire soit une erreur.

# Grammaire LL(1)

- **Algorithme d'analyse prédictive :** L'algorithme permettant de faire l'analyse syntaxique prédictive est le suivant :

```
positionner le pointeur pt sur le premier symbole de w$
répéter
soit X le symbole en sommet de la pile et a le symbole repéré par
pt
   si X est un terminal ou $ alors
       si X = a alors
          enlever X de la pile, avancer pt
       sinon si X = $ alors succès
             sinon alors erreur
   sinon
       si M(X,a)= X → Y1Y2... Yk alors
          début
               enlever X de la pile
               mettre  Y1Y2... Yk avec Y1 au sommet
           fin
       sinon erreur
  jusqu'à X = $ (pile vide)
```

# Grammaire LL(1)

- **Algorithme d'analyse prédictive :**

**Exemple** : Soit la chaîne w= id+id*id et la grammaire suivante :

$E \rightarrow TE'$

$E' \rightarrow +TE' \mid \varepsilon$

$T \rightarrow FT'$

$T' \rightarrow *FT' \mid \varepsilon$

$F \rightarrow (E) \mid id$

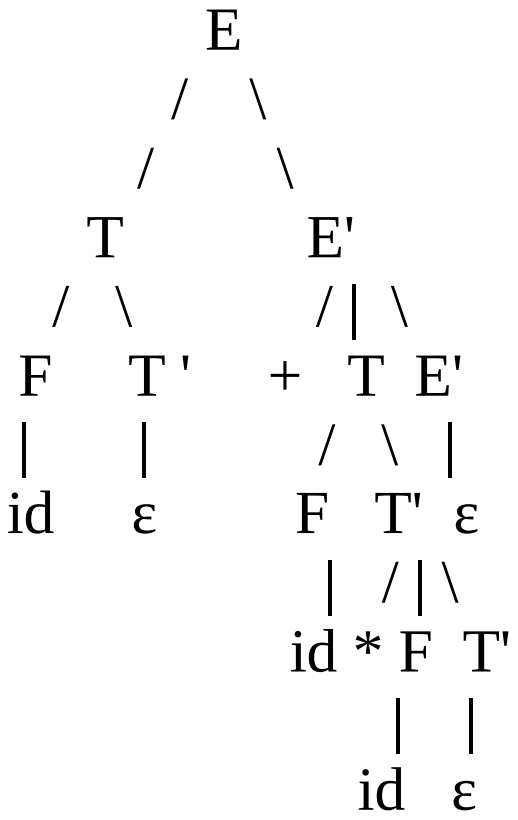
# Grammaire LL(1)

Table Analyse

| Épile | Pile | Entrée | Action |
|---|---|---|---|
| | E$ | id+id*id$ | E → TE' |
| | TE'$ | id+id*id$ | T → FT' |
| | FT'E'$ | id+id*id$ | F → id |
| | id T'E'$ | id+id*id$ | Dépiler id |
| id | T'E'$ | +id*id$ | T' → ε |
| id | E'$ | +id*id$ | E' → +TE' |
| id | +TE'$ | +id*id$ | Dépiler + |
| id+ | TE'$ | id*id$ | T → FT' |
| id+ | FT'E'$ | id*id$ | F → id |
| id+ | idT'E'$ | id*id$ | Déplier id |
| id+id | T'E'$ | *id$ | T' → *FT' |
| id+id | *FT'E'$ | *id$ | Dépiler * |
| id+id* | FT'E' | id$ | F → id |
| id+id* | idT'E'$ | id$ | Dépiler id |
| id+id*id | T'E'$ | $ | T' → ε |
| id+id*id | E'$ | $ | E' → ε |
| id+id*id | $ | $ | Accepter |

# Grammaire LL(1)

- **Algorithme d'analyse prédictive :** L'arbre correspondant :




**Exercice d'application :** appliquer l'algorithme sur la chaîne (a+b) * (c + d) avec a,b,c,d sont des identificateurs. Donner l'arbre syntaxique correspondant.

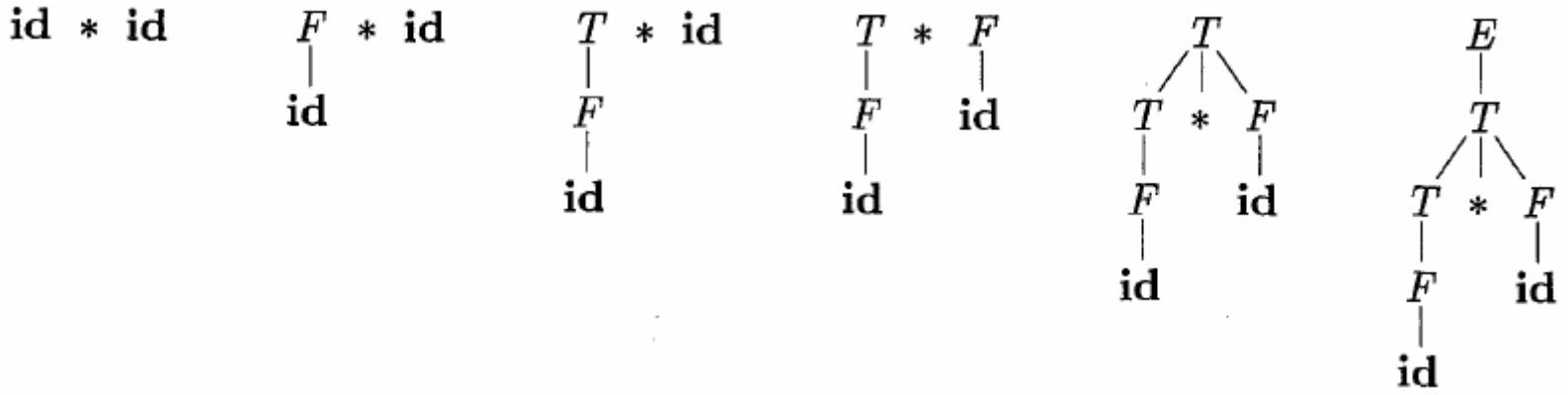
# L'analyse ascendante

- **Principe** : construire un arbre de dérivation du bas (les feuilles, c-à-d les unités lexicales) vers le haut (la racine, c-à-d l'axiome de départ).
  - **Exemple** : w = id * id

    E → E+T|T

    T → T*F|F

    F →( E ) | id



# L’analyse ascendante

- 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

# L’analyse ascendante

- Le modèle décalages-réductions est une analyse dans laquelle une pile contient les symboles de grammaire et un tampon d'entrée contient le reste de la chaîne à analyser.
- $ est utilisé pour marquer le bas de la pile et aussi l'extrémité droite de l'entrée. Classiquement, lorsqu'il est question de l'analyse ascendante, nous montrons le haut de la pile sur la droite, plutôt que sur la gauche comme nous l'avons fait pour l'analyse descendante. Au départ, la pile est vide, et la chaîne w est sur l'entrée, comme suit :

| | Pile | Entrée |
|---|---|---|
| état initial | $ | w$ |
| état final | $S | $ |

# L’analyse ascendante

- Les opérations permises sont :
  - **Décalage** : le symbole d'entrée courant est inséré dans la pile,
  - **Réduction** : on reconnaît sur le sommet de la pile une partie droite d’une production $Y \rightarrow X_{k-j}...X_j$, on l’enlève et on la remplace par sa partie gauche Y
  - **Acceptation** : Annoncer la réussite de l'analyse.
  - **Erreur** : Découverte d'une erreur de syntaxe ou appel d'une routine de récupération d'erreur.

# L’analyse ascendante

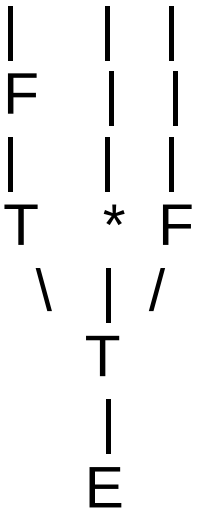
- **Exemple** : w = id1 * id2

E → E+T|T

T → T*F|F

F →( E ) | id

Arbre syntaxique




| Pile | Entrée | Action |
|---|---|---|
| $ | id1*id2$ | Décalage |
| $id1 | *id2$ | Réduction par F → id |
| $F | *id2$ | Réduction par T → F |
| $T | *id2$ | Décalage |
| $T* | id2$ | Décalage |
| $T*id2 | $ | Réduction par F → id |
| $T*F | $ | Réduction par T → T*F |
| $T | $ | Réduction par E → T |
| $E | $ | Accepter |

# L’analyse LR

- L'analyse LR(k) est l'un des analyseurs les plus utilisés des analyses ascendantes, L signifie analyse de gauche à droite (Left to right) de l'entrée, le R pour la construction des dérivations à droite en inverse, et le k pour le nombres des symboles d'entrée dans la recherche pour les décisions d'analyse.
- On distingue différents types d'analyse LR : SLR (Simple LR), LR canonique (Canonical LR) et LALR (Look Ahead LR).
- Les générateurs d'analyseurs syntaxiques implémentent cette méthode (spécialement LALR pour Yacc).

# L'analyse SLR

- **Les items d'un automate LR(0) :**
  - **Question :** comment savoir quand décaler et quand réduire ?
  - L'analyseur LR effectue les décisions de décalage-réduction par le maintient des états d'analyse pour savoir l'endroit où on est dans l'analyse. Les états représentent un ensemble d'items.
  - Un item LR(0), (ou item) d'une grammaire G est une production de G avec un point repérant une position de sa partie droite.
  - **Exemple** : la production A → XYZ fournit 4 items :
    - A → ·XYZ  A → X·YZ  A → XY·Z  A → XYZ·
  - La production A →ε fournit uniquement l'item A → ·

# L'analyse SLR

- **Les items d'un automate LR(0) :**
  - Un item peut être codé par un couple d'entiers, le premier pour le numéro de la production, le second pour la position du point.
  - Intuitivement, un item indique la « quantité » de la partie droite qui a été reconnue à un moment donné au cours du processus d'analyse.
  - Par exemple le premier item ($A \rightarrow \cdot XYZ$ ) indique que l'on espère voir en entrée une chaîne dérivable depuis XYZ.
  - Le second item ($A \rightarrow X \cdot YZ$ ) indique que nous venons de voir en entrée une chaîne dérivée de X et que nous espérons maintenant voir une chaîne dérivée de YZ.

# L’analyse SLR

- **Idée :** L’idée de base de la méthode SLR est de construire tout d’abord, à partir de la grammaire, un AFD pour prendre les décisions d'analyse. Cet automate est appelé automate LR(0). Les items sont regroupés en ensembles qui constituent les états de l’analyseur SLR.
  - Une collection d’ensembles d’items LR(0) que nous appellerons collection LR(0) canonique, fournit la base de la construction des analyseurs SLR.
  - Pour construire la collection LR(0) canonique pour une grammaire, nous définissons une grammaire augmentée et deux fonctions : Fermeture et Transition.
  - Si G est une grammaire d’axiome S, alors la grammaire augmentée de G est G' avec un nouvel axiome S' et une nouvelle production S' → S.
  - Le but de cette nouvelle production initiale est d’indiquer à l’analyseur quand il doit arrêter l’analyse et annoncer que la chaîne d’entrée a été acceptée. Plus précisément la chaîne d’entrée est acceptée quand et seulement quand l’analyseur est sur le point de réduire S' → S.

# L'analyse SLR

- **Fermeture (closure) :** Si I est un ensemble d'items pour une grammaire G, Fermeture(I) est l'ensemble d'items construit à partir de I par les deux règles :
  1) Initialement, placer chaque item de I dans Fermeture(I).
  2) Si $A \rightarrow \alpha \cdot B\beta$ est dans Fermeture(I) et $B \rightarrow \gamma$ est une production, ajouter l'item $B \rightarrow \cdot\gamma$ à Fermeture(I) s'il ne s'y trouve pas déjà.
  - Nous appliquons cette règle jusqu'à ce qu'aucun nouvel item ne puisse plus être ajouté à Fermeture(I).
- Intuitivement, si $A \rightarrow \alpha \cdot B\beta$ est dans Fermeture(I) cela indique qu'à un certain point du processus d'analyse, nous pourrions voir se présenter dans l'entrée une sous-chaîne dérivable depuis Bβ comme entrée.
- Si $B \rightarrow \gamma$ est une production, nous supposons que nous pourrions également voir, à ce moment là, une chaîne dérivable depuis γ.
- Pour cette raison, nous ajoutons également $B \rightarrow \cdot\gamma$ à Fermeture(I ).

# L’analyse SLR

- **Algorithme :**

```
SetOfItems Fermeture(I){
    J = I ;
    repeat
        for(chaque item A → α·Bβ dans J)
            for(chaque production B → γ dans G)
                si(B → ·γ n'est pas encore dans J)
                    ajouter  B → ·γ à J ;
    jusqu'à ce qu'aucun item ne peut plus être ajouter à J ;
    return J ;
}
```

# L'analyse SLR

- **Exemple** : soit la grammaire E' → E ; E → E + T | T ;
  T → T * F | F ; F → (E) | id
- Si I = {[E' → ·E]}, alors Fermeture(I) contient les items :
  Fermeture(I)={E' → ·E,E → ·E + T,E → ·T,T → ·T * F,
  T → ·F, F → ·(E), F → ·id}

**Explication** : conformément à la première règle, E' → ·E est dans Fermeture(I). puisqu'il existe un E immédiatement à droite du point, nous ajoutons les productions de E avec des points dans la fin gauche : E → ·E+T et E → ·T. Comme T est immédiatement dans la droite du point du dernier item, donc ajouter T → ·T*F et T → ·F. Par la suite F se trouve à droite du point donc ajouter F → ·(E) et F → ·id.

# L'analyse SLR

- **Transition (goto) :** La deuxième fonction utile est Transition(I,X) où I est un ensemble d'items et X est un symbole de la grammaire.
- Transition(I,X) est définie comme la fermeture de l'ensemble de tous les items [A → αX·β] tels que [A → α·Xβ] appartient à I
- **Exemple** : Si I = {[E → E·], [E → E·+T]}, alors Transition(I ,+) consiste en :

E → E +·T
T → ·T * F
T → ·F
F → ·(E)
F → ·id

**Explication** : Nous calculons Transition(I,+) en cherchant dans I les items ayant + immédiatement à droite du point. E → E · ne convient pas, E → E·+T répond au critère. Nous faisons franchir le + au point afin d'obtenir {[E → E + ·T ]}, puis nous calculons Fermeture sur cet ensemble.

# L'analyse SLR

- **Algorithme de construction des items :**
- **Donnée :** grammaire augmentée G'
- **Résultats :** une collection canonique d'ensembles d'items LR(0) C pour la grammaire G'
- **Méthode :** exécuter les instructions suivantes pour ajouter les transitions sur tous les symboles à partir de l'axiome.

```
void items(G'){
C = fermetrure({[S' →·S]} ;
      repeat
            for(chaque ensemble d'items I dans C)
                for(chaque symbole X de la grammaire)
                    si(Transition(I,X) est non vide et n'est pas encore dans C)
                      ajouter Transition(I,X) à C ;
    until aucun nouvel ensemble d'items ne peut plus être ajouté à C
}
```

# L'analyse SLR

- **Algorithme de construction des items :**
- **Donnée :** soit la grammaire E' → E ; E → E + T | T ; T → T * F | F ; F → (E ) | id
- Initialiser C = Fermeture({[E' → E]})
- Soit I0 = Fermeture({[E' → E]})={[E' → ·E], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- I1=Transition(I0,E)={[E' → E·], [E → E·+T]}
- I2=Transition(I0,T)={[E → T·], [T → T·*F]}
- I3=Transition(I0,F)={[T → F·]}
- I4=Transition(I0,()={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}

# L'analyse SLR

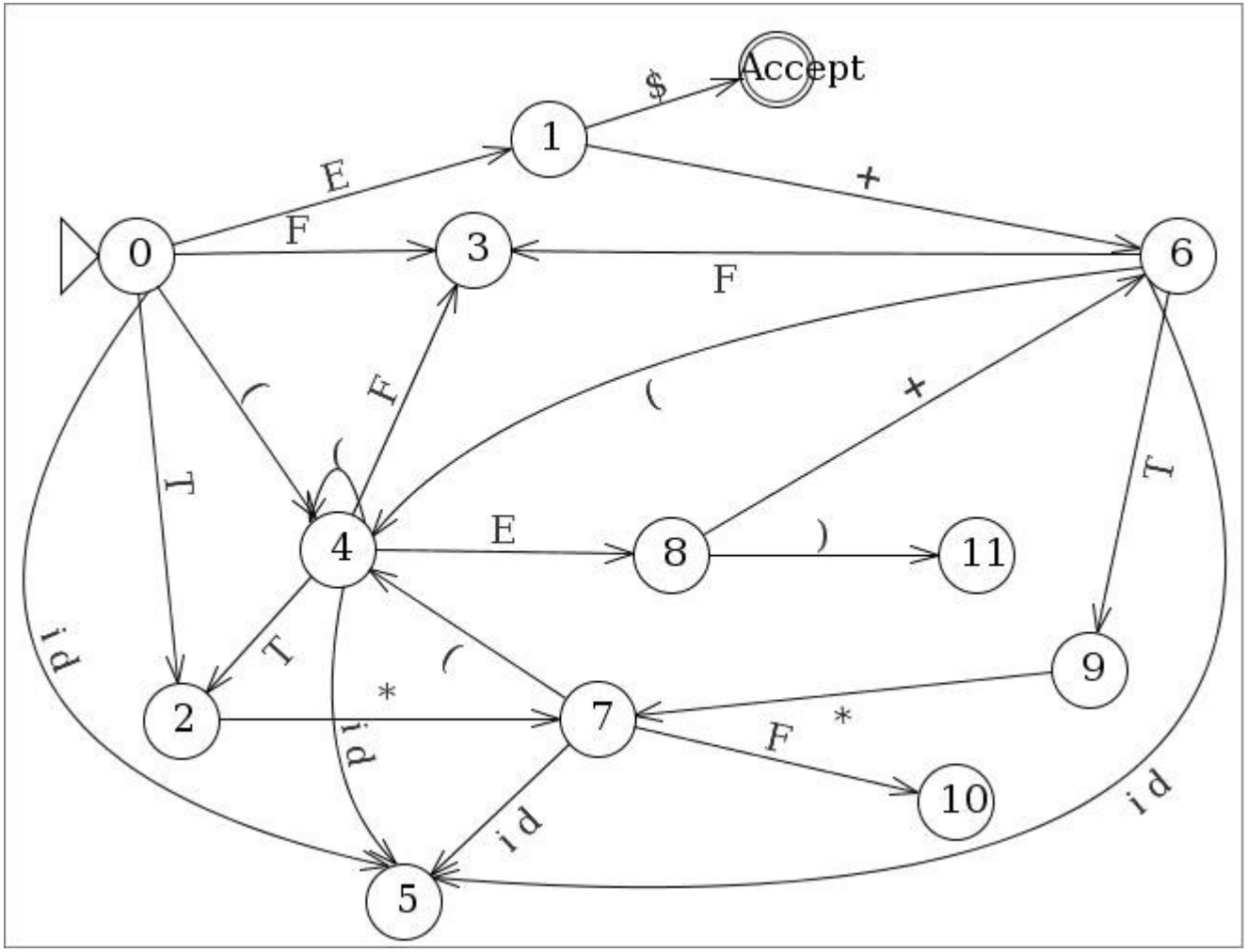
- **Algorithme de construction des items :**
- **Donnée :** soit la grammaire E' → E ; E → E + T | T ; T → T * F | F ; F → (E ) | id
- I5=Transition(I0,id)={[F → id·]}
- I6=Transition(I1,+)={[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]}
- I7=Transition(I2,*)={[T → T*·F], [F → ·(E)], [F → ·id]}
- I8=Transition(I4,E)={[F → (E·)], [E → E·+T]}
- Transition(I4,T)={E → T·], [T →T· * F]}=I2
- Transition(I4,F)={[T → F·]}=I3
- Transition(I4,()={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}=I4
- Transition(I4,id)={[F → id·]}=I5
- I9=Transition(I6,T)={[E → E+T·], [T → T· * F]}
- Transition(I6,F)={[T → F·]}=I3
- Transition(I6,()={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}=I4
- Transition(I6,id)={[F → id·]}=I5

# L'analyse SLR

- **Algorithme de construction des items :**
- **Donnée :** soit la grammaire E' → E ; E → E + T | T ; T → T * F | F ; F → (E ) | id
- I10=Transition(I7,F)={[T → T*F·]}
- Transition(I7,()={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}=I4
- Transition(I7,id)={[F → id·]}=I5
- I11=Transition(I8,))={[F → (E)·]}
- Transition(I8,+)=[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]=I6
- Transition(I9,*)={[T → T*·F], [F → ·(E)], [F → ·id]}=I7

# L'analyse SLR

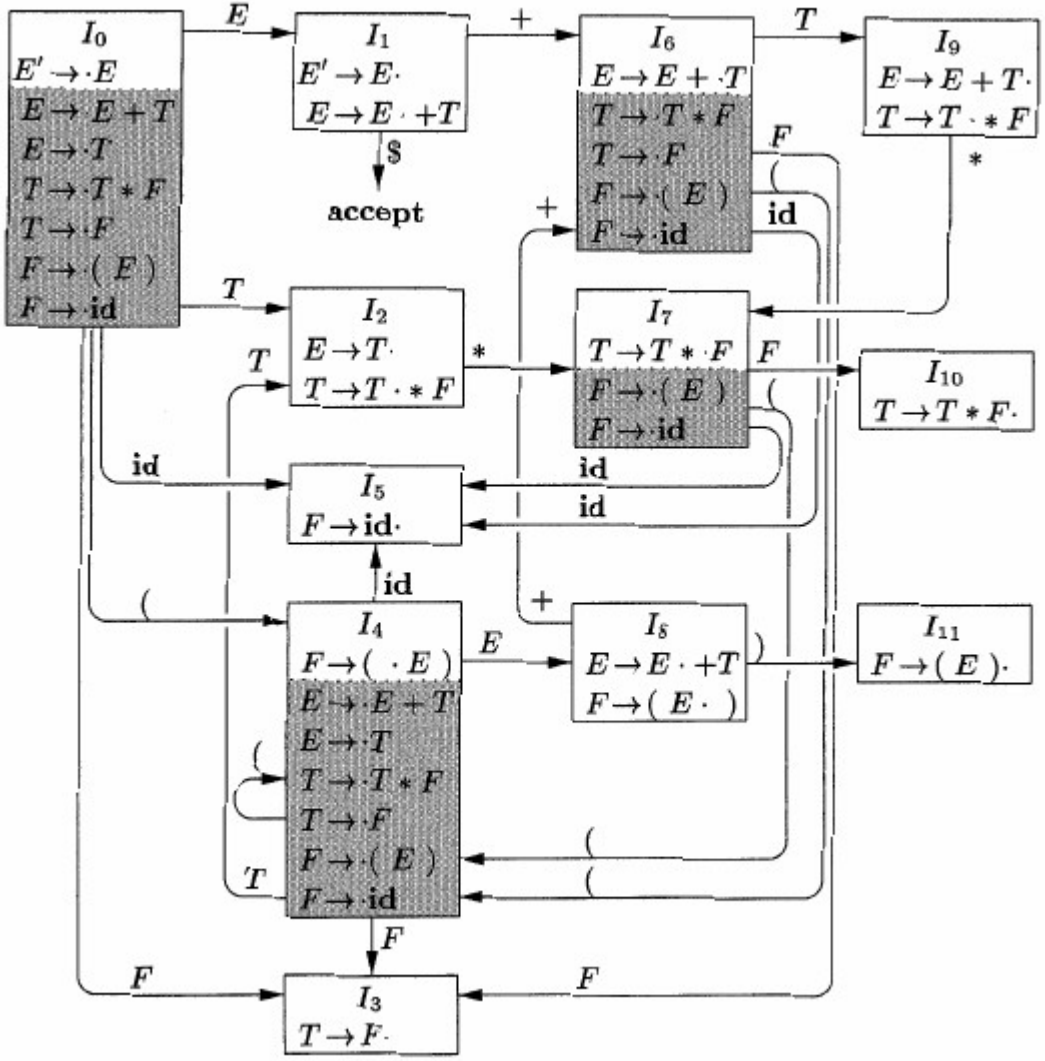
- **L'AFD construit pour la fonction Transition :**




L'état de départ de l'automate LR(0) est Fermeture({[S' → ·S]}), avec S' le symbole de départ de la grammaire augmentée. Tous les états sont des états d'acceptation. Les décisions de décalage-réduction peuvent se faire comme suit. Supposons que la chaîne γ formée des symboles de la grammaire parcourt l'AFD LR(0) depuis l'état de départ 0 vers un état j. Alors décaler vers un symbole a si l'état j a une transition sur a. sinon, on choisit réduire, les items dans l'état j indiqueront quelle production à utiliser.

# L'analyse SLR

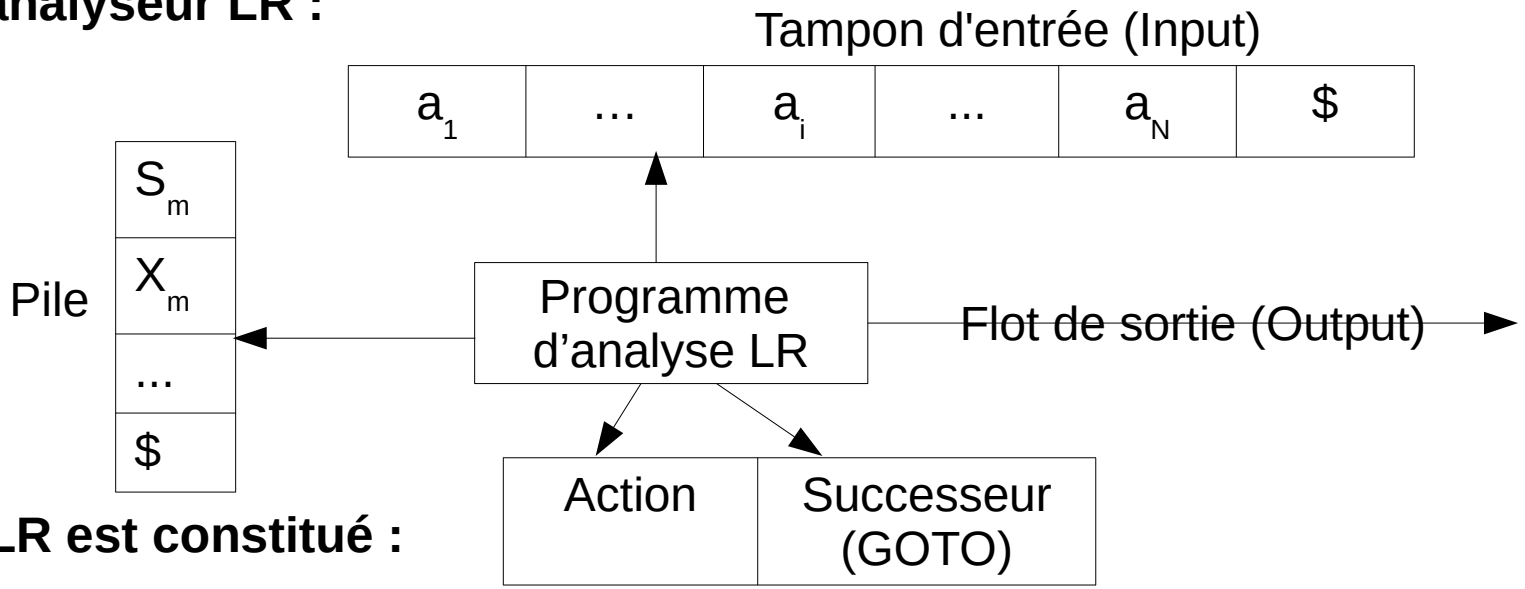
- **L'AFD construit pour la fonction Transition :**




AFD avec liste des items de chaque état

# L'algorithme d'analyse LR

- **Schéma d'un analyseur LR :**




- **Un analyseur LR est constitué :**
  - D'un tampon d'entrée
  - D'un flot de sortie
  - D'une pile
  - D'un programme pilote
  - Des tables d'analyse divisées en deux parties (Action et Successeur).
- **Remarque :** Le programme pilote d'analyse LR est le même pour tous les analyseurs LR (SLR, LR canonique, LALR), seul les tables d'analyse changent d'un analyse à un autre.

# L'algorithme d'analyse LR

- **Structure de la table d'analyse LR :**
- La pile contient une séquence d'états, $s_0 s_1 ... s_m$, où $s_m$ est au sommet de la pile.
- Dans la méthode SLR, la pile contient les états de l'AFD LR(0) ; les méthodes LR canonique et LALR sont similaires.
- La table d'analyse consiste en deux parties : la fonction Action d'analyse et la fonction Successeur.
- La fonction Action prend comme arguments un état i et un terminale a (ou $ le marqueur de fin). La valeur de Action[i,a] peut prendre 4 formes :

  a) décaler vers j, où j est un état. L'action prise par l'analyseur décale a dans la pile, mais il utilise l'état j à ça place pour représenter a.

  b) réduire $A \rightarrow \beta$. L'action de réduire β par la tête A.

  c) accepter

  d) erreur

- La fonction Successeur est la même que la fonction Transition définie pour les états des items, si Transition[$I_i$,A] = $I_j$ alors Successeur[i,A]=j, avec i,j des états de l'AFD et A un non terminale.

# L'algorithme d'analyse LR

- **Comportement d'un analyseur LR :**
- Une configuration d'un analyseur LR est un couple (pile, tampon d'entrée) dont le premier composant est le contenu de la pile et dont le second composant est la chaîne d'entrée restant à analyser : ($S_0$ $X_1S_1X_2S_2$ . . . $X_mS_m$, $a_ia_{i+1}$ . . . $a_n$\$) qui représente la proto-phrase droite $X_1X_2$. . .$X_ma_ia_{i+1}$ . . .$a_n$\$ (avec $S_i$ est un état représentant $X_i$ un symbole de la grammaire).
- L'action suivante de l'analyseur est de déterminer par la lecture de $a_i$, le symbole d'entrée courant, et $S_m$, l'état au sommet de la pile, ensuite consulter l'entrée Action[$S_m$,$a_i$] dans la table des actions. Les configurations résultantes après chacun des quatre types d'action sont :
  - Si Action[$S_m$,$a_i$]=décaler S, l'analyseur exécute une action décaler atteignant la configuration ($S_0X_1S_1X_2S_2$ . . . $X_mS_m$ $a_i$ S, $a_{i+1}$ . . . $a_n$\$). L'analyseur a à la fois empilé le symbole d'entrée courant $a_i$ et le prochain état S, qui est donné par Action[$S_m$,$a_i$], $a_{i+1}$ devient le symbole d'entrée courant.

# L'algorithme d'analyse LR

- **Comportement d'un analyseur LR :**
  - Si Action[$S_m$,$a_i$]=réduire par A → β, alors l'analyseur exécute une action réduire, atteignant la configuration : ($S_0 S_1 S_2 \ldots S_{m-r} A S$, $a_i a_{i+1} \ldots a_n$\$), avec r est la longueur de β et S=Successeur[$S_{m-r}$ ,A].
    - L'analyseur ici dépile r états et r symboles, et met ainsi l'état $S_{m-r}$ au sommet de la pile.
    - L'analyseur empile alors S=Successeur[$S_{m-r}$,A]. le symbole courant en entrée n'est pas changé par une action réduire.
    - Pour les analyseurs LR, $X_{m-r+1} \ldots X_m$, la séquence de symboles grammaticaux correspondant aux dépilés, correspondra toujours à β, le symbole à droite de la production réduite.
    - Le résultat d'un analyseur LR est généré après une action réduire en exécutant une action sémantique associée à la production considérée.
    - Dans notre cas on supposera que la sortie est uniquement formée par la liste des productions par lesquelles on a réduit.

# L’algorithme d'analyse LR

- **Comportement d'un analyseur LR :**
  - Si Action[$S_m$,$a_i$] = accepter, l’analyseur a terminé.
  - Si Action[$S_m$,$a_i$] = erreur, l’analyseur a découvert une erreur et appelle une routine de récupération sur erreur.
  - L’algorithme d’analyse LR est résumé dans le diapos ( 73) suivant.
  - Tous les analyseurs LR se comportent de la même façon la seule différence entre deux analyseurs LR est dans les champs Action et Successeur de la table d'analyse.

# L’algorithme d'analyse LR

- **Algorithme d'un analyseur LR :**
  - **Donnée** : une chaîne d’entrée w avec une table d’analyse LR et les fonctions Action et Successeur pour une grammaire G.
  - **Résultat** : si w est dans L(G), une analyse ascendante de w (étapes de réductions) ; sinon une indication d’erreur.
  - **Méthode** : initialement, l’analyseur a $S_0$ en sa pile, où $S_0$ est l’état initial, et w$ dans son tampon d’entrée. L’analyseur exécute le programme ci-après (diapo 74) jusqu’à ce qu’il rencontre soit une action accepter soit une action erreur.

# L'algorithme d'analyse LR

- **Algorithme d'un analyseur LR :**

```
Considérer a comme étant le premier symbole de w$
while(1){/*répéter inifiniment*/
 considérer S l'état au sommet de la pile,
 if(Action[S,a] = décaler t){
   empiler t et a ; /*mettre t et a dans la pile*/
   a = symbole suivant ;
}else if(Action[S,a]=réduire A → β){
   Dépiler β de la pile ;
   t = nouveau sommet de la pile;
   Empiler A et Successeur[t,A] dans la pile ;
   afficher la production A → β ;
}else if(Action[S,a]=accept) break ; /*analyse
terminé*/
      else appel routine de traitement d'erreur ;
}
```

# L'algorithme d'analyse LR

- **Algorithme d'un analyseur LR : exemple : soit la grammaire suivante**

  (1) E → E + T

  (2) E → T

  (3) T → T * F

  (4) T → F

  (5) F → (E)

  (6) F → id

- **Le codage des actions et le suivant :**
  - di signifie décaler et empiler l'état i.
  - rj signifie réduire par la production j (ici de 1 à 6).
  - acc signifie accepter.
  - une entrée vide signifie erreur.

# L'algorithme d'analyse LR

- **Algorithme d'un analyseur LR : exemple :**

(1) E → E + T

(2) E → T

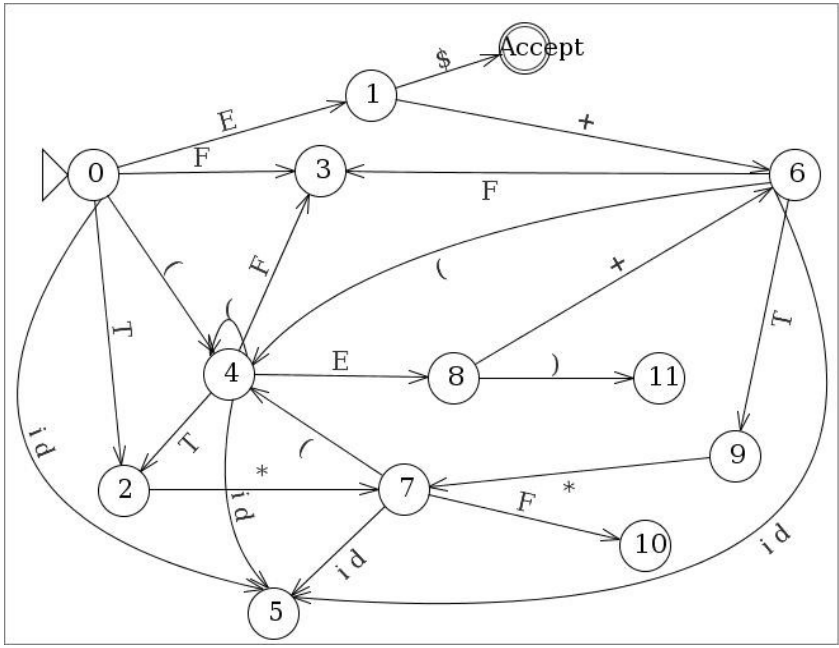
(3) T → T * F

(4) T → F

(5) F → (E)

(6) F → id

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |



# L'algorithme d'analyse LR

- **Algorithme d'un analyseur LR : exemple : sur l'entrée id*id+id la pile et le tampon sont :**

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |

| Pile | Tampon d'entrée | Action |
|---|---|---|
| (1) 0 | id*id+id$ | d5 |
| (2) 0 id 5 | *id+id$ | r par F → id |
| (3) 0 F 3 | *id+id$ | r par T → F |
| (4) 0 T 2 | *id+id$ | d7 |
| (5) 0 T 2 * 7 | id+id$ | d5 |
| (6) 0 T 2 * 7 id 5 | +id$ | r par F → id |
| (7) 0 T 2 * 7 F 10 | +id$ | r par T → T * F |
| (8) 0 T 2 | +id$ | r par E → T |
| (9) 0 E 1 | +id$ | d6 |
| (10) 0 E 1 + 6 | id$ | d5 |
| (11) 0 E 1 + 6 id 5 | $ | r par F → id |
| (12) 0 E 1 + 6 F 3 | $ | r par T → F |
| (13) 0 E 1 + 6 T 9 | $ | r par E → E + T |
| (14) 0 E 1 | $ | accepter |

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

# Construction de la table d'analyse SLR

- La méthode SLR commence par les items LR(0) et l'automate LR(0).
- On augmente la grammaire G en G', avec un nouveau axiome S'.
- On construit C la collection canonique des ensembles d'items pour G' avec leur fonction de transition.
- Les entrées Action et Successeur dans la table d'analyse sont construites en utilisant l'algorithme suivant (diapo 79).
- Il est nécessaire de savoir suivant(A) pour chaque non-terminal de la grammaire.

# Construction de la table d'analyse SLR

- **Algorithme** : construction table SLR.
- **Donnée** : une grammaire augmentée G'.
- **Résultat** : table d'analyse SLR des fonctions Action et Successeur pour G'.
- **Méthode :**

1) Construire la collection C ={$I_0$,$I_1$,...,$I_n$} des ensemble des items de LR(0) pour G'.
2) l'état i est construit à partir de $I_i$, Les actions d'analyse pour l'état i sont déterminées comme suit :

   (a) Si [A → α·aβ] (a doit être un terminal) est dans $I_i$ et Transition($I_i$,a)=$I_j$, donc Action[i,a]=décaler j.

   (b) Si [A → α·] (A ≠ S') est dans $I_i$ , donc Action[i,a]=réduire A → α pour chaque a dans Suivant(A).

   (c ) Si [A → S·] est dans $I_i$ , donc Action[i,$]=accepter.

- Si les règles précédentes engendrent des actions conflictuelles, nous disons que la grammaire n'est pas SLR(1). Dans ce cas, l'algorithme échoue et ne produit pas d'analyseur.

# Construction de la table d'analyse SLR

- **Algorithme** : construction de la table SLR.
- **Méthode :**

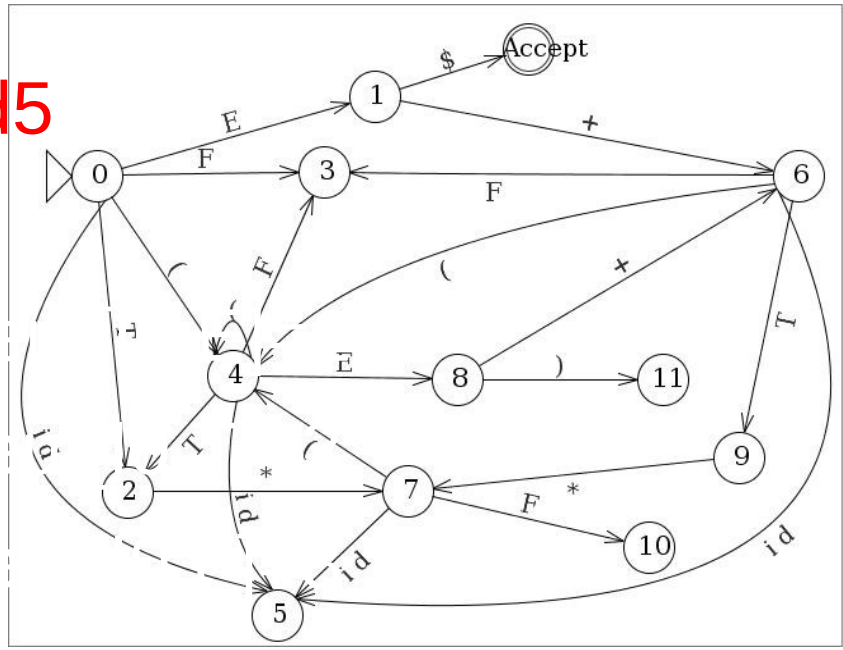
1) ...
2) ...
3) On construit les transitions Successeurs pour l'état i pour tout non-terminal A en utilisant la règle : si Transition($I_i$,A) = $I_j$ alors Successeur[i, A] = j.
4) Toutes les entrées non définies par les règles 2 et 3 sont positionnées à « erreur ».
5) L'état initial de l'analyseur est celui qui est construit à partir de l'ensemble d'items contenant [S' → ·S].

# Construction de la table d'analyse SLR

- La table d'analyse des fonctions Action et Successeur déterminée par l'algorithme précédent est appelée table SLR(1) pour G.
- Un analyseur LR qui utilise la table SLR(1) pour G est appelé un analyseur SLR(1) pour G, et une grammaire ayant une table SLR(1) est dite SLR(1).
- Habituellement on omit le (1) après SLR puisque ne nous traitons pas les analyseurs qui acceptent plus d'un symbole en entrée.

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I0={[E' → ·E],[E → ·E+T|·T], [T →·T * F|·F], [F → ·(E)|·id]}
- Transition(I0,id)=I5 donc A[0,id]=d5

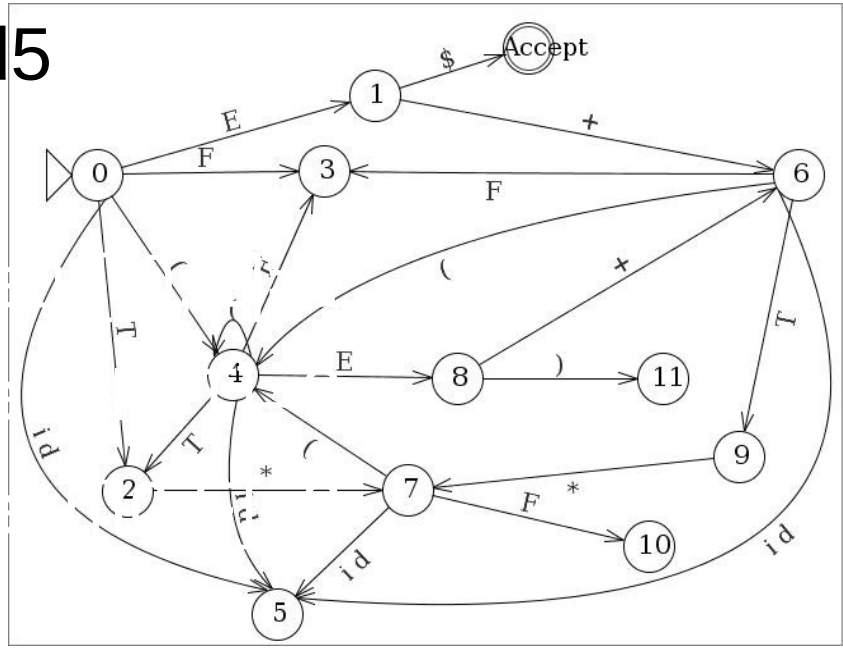



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I0={[E' → ·E],[E → ·E+T|·T], [T →·T * F|·F], [F → ·(E)|·id]}
- Transition(I0,id)=I5 donc A[0,id]=d5
- Transition[I0,(]=I4 donc A[0,(]=d4

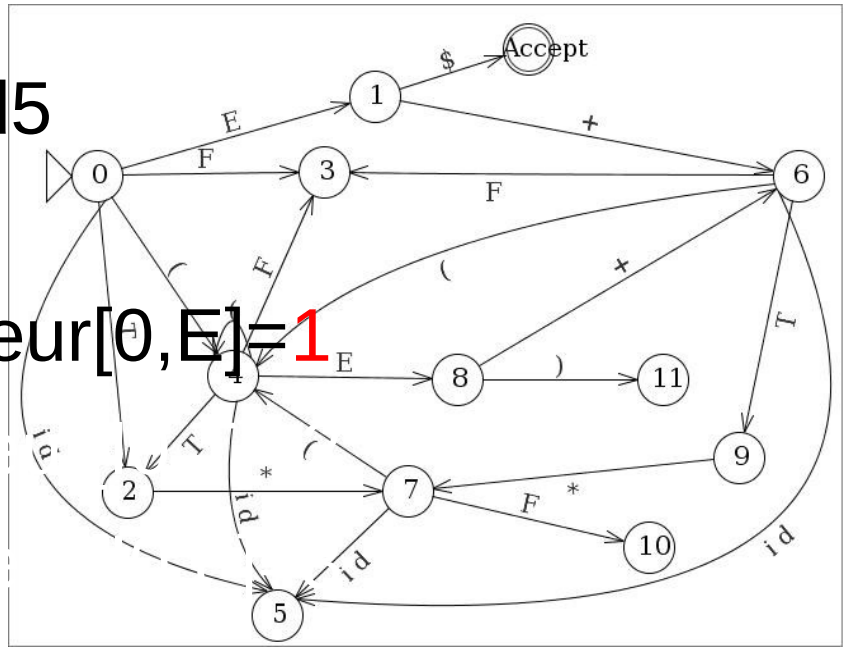



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I0={[E' → ·E],[E → ·E+T|·T], [T →·T * F|·F], [F → ·(E)|·id]}
- Transition(I0,id)=I5 donc A[0,id]=d5
- Transition[I0,(]=I4 donc A[0,(]=d4
- Transition[I0,E]=I1 donc Successeur[0,E]=1

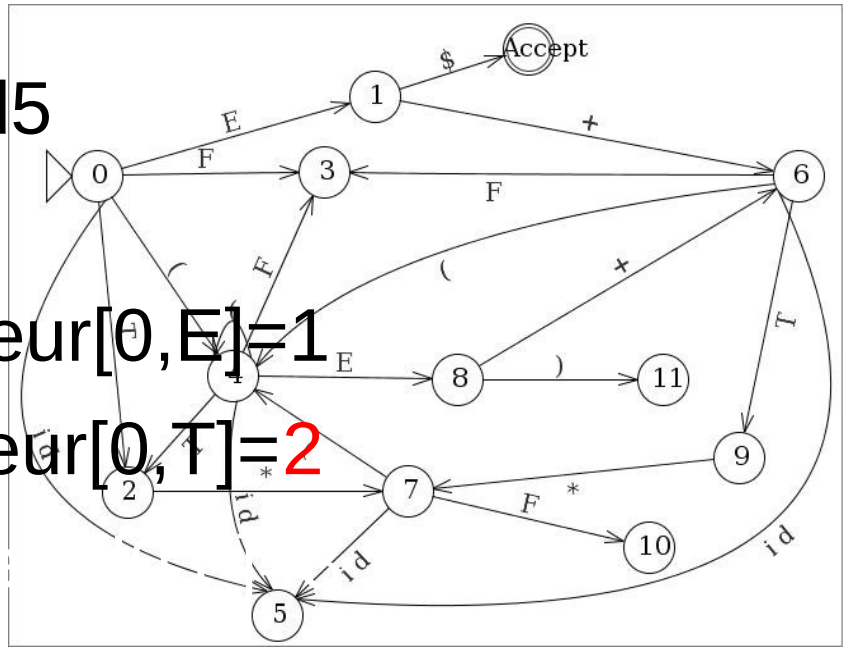



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I0={[E' → ·E],[E → ·E+T|·T], [T →·T * F|·F], [F → ·(E)|·id]}
- Transition(I0,id)=I5 donc A[0,id]=d5
- Transition[I0,(]=I4 donc A[0,(]=d4
- Transition[I0,E]=I1 donc Successeur[0,E]=1
- Transition[I0,T]=I2 donc Successeur[0,T]=2

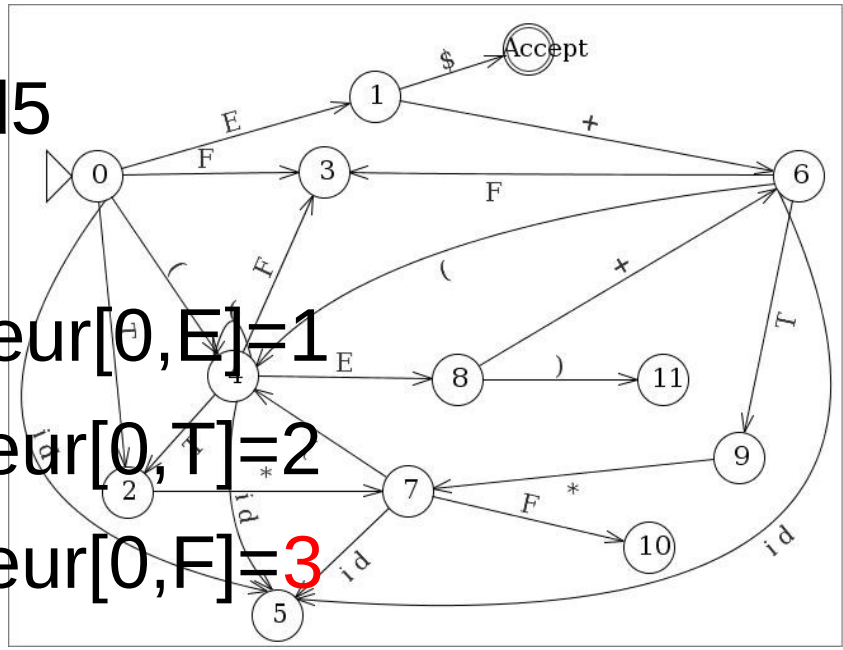



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I0={[E' → ·E],[E → ·E+T|·T], [T →·T * F|·F], [F → ·(E)|·id]}
- Transition(I0,id)=I5 donc A[0,id]=d5
- Transition[I0,(]=I4 donc A[0,(]=d4
- Transition[I0,E]=I1 donc Successeur[0,E]=1
- Transition[I0,T]=I2 donc Successeur[0,T]=2
- Transition[I0,F]=I3 donc Successeur[0,F]=3

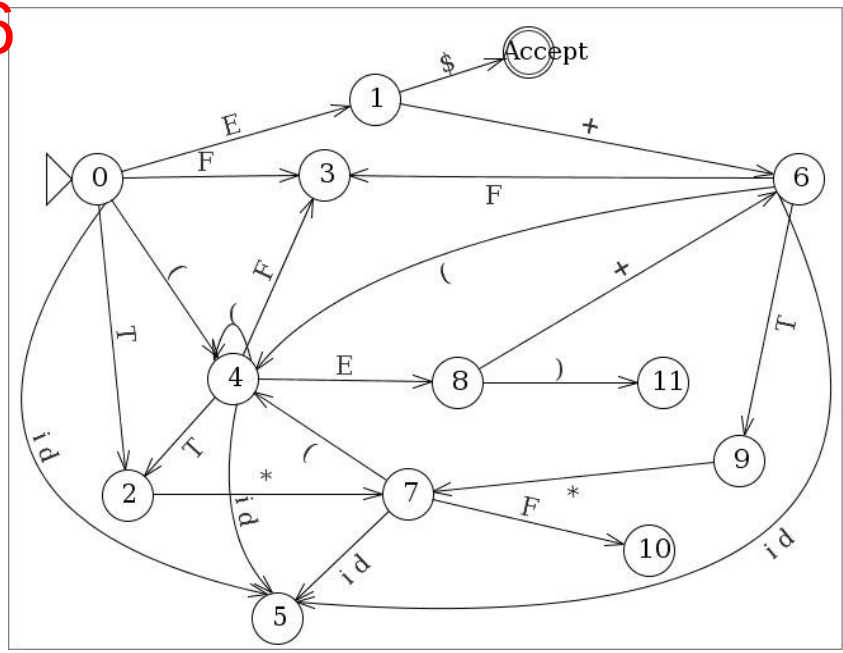



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I1={[E' → E·], [E → E·+T]}
- Transition[I1,+]=I6 donc A[1,+]=d6

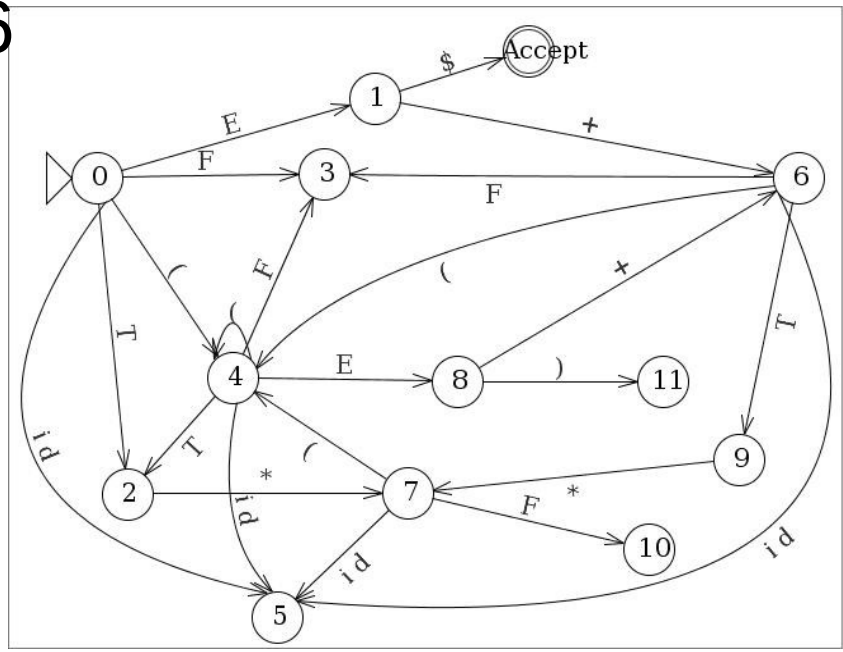



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I1={[E' → E·], [E → E·+T]}
- Transition[I1,+]=I6 donc A[1,+]=d6
- E' →E· donc A[1,$]=accepte

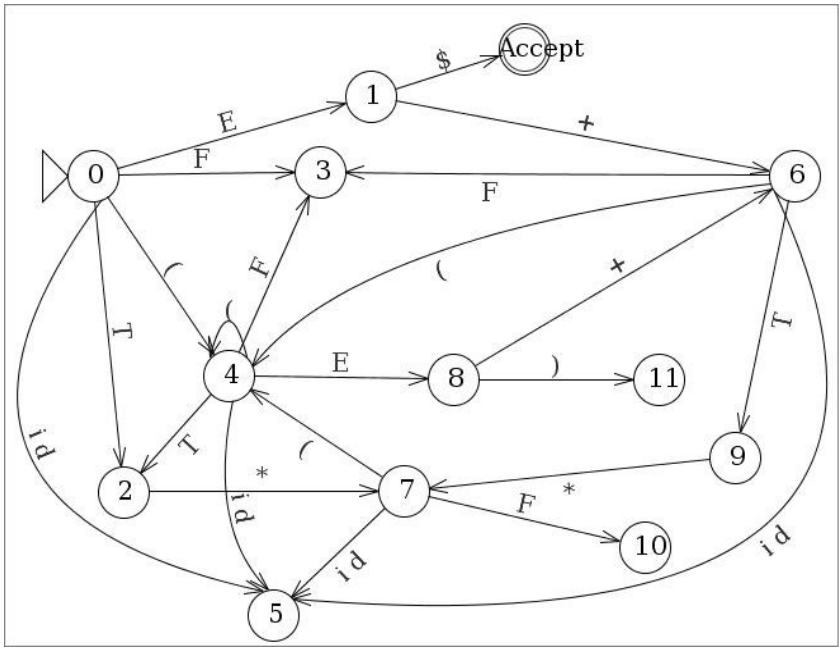



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I2={[E → T·], [T → T·*F]}
- Transition[I2,*]=I7 donc A[2,*]=d7

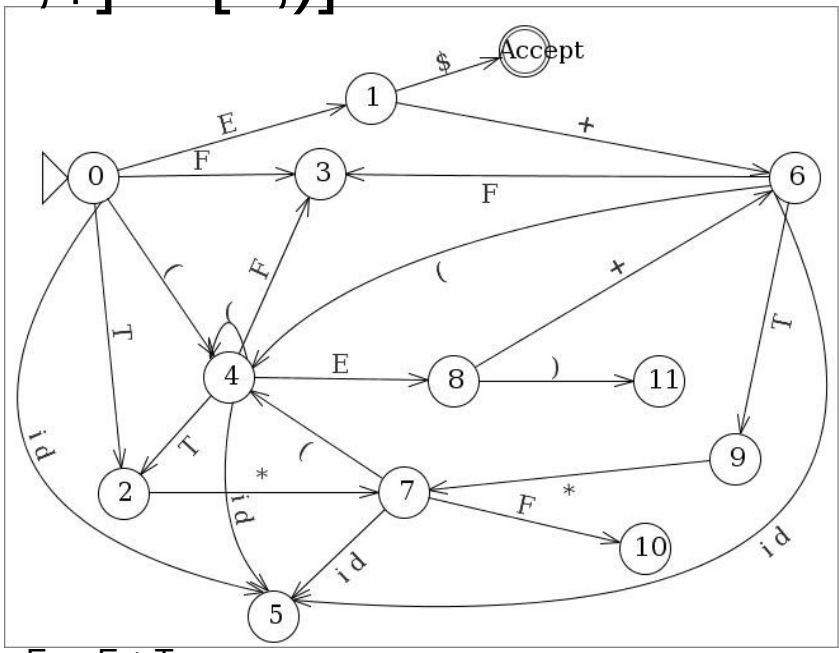



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | | d7 | | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I2={[E → T·], [T → T·*F]}
- Transition[I2,*]=I7 donc A[2,*]=d7
- Suivant(E)={$,+,)} alors A[2,+]=A[2,$]=A[2,)]=réduire/2



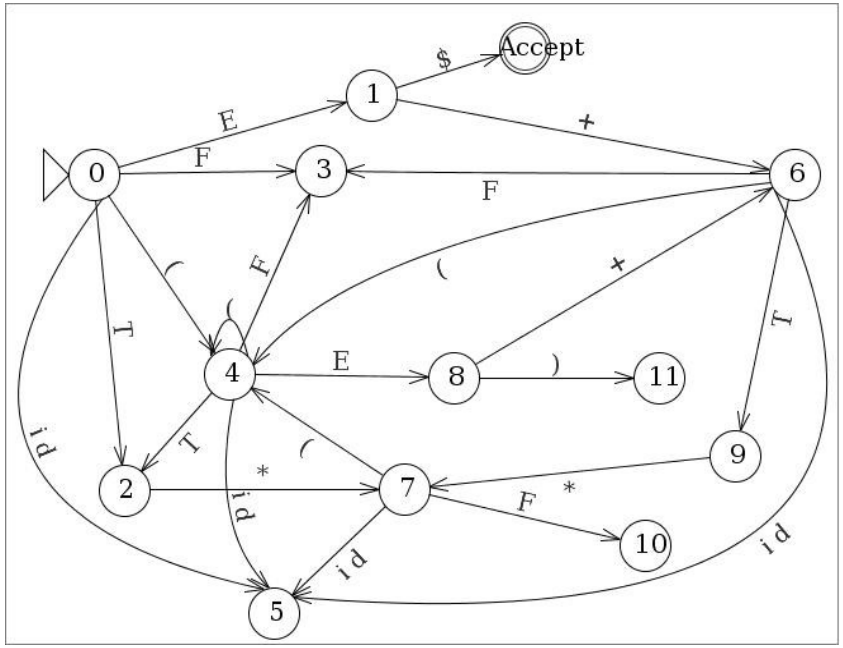

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I3={[T → F·]}
- Suivant(T)={$,+,*,)} alors A[3,$]=A[3,*]=A[3,)]=A[3,+]= réduire/4

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |

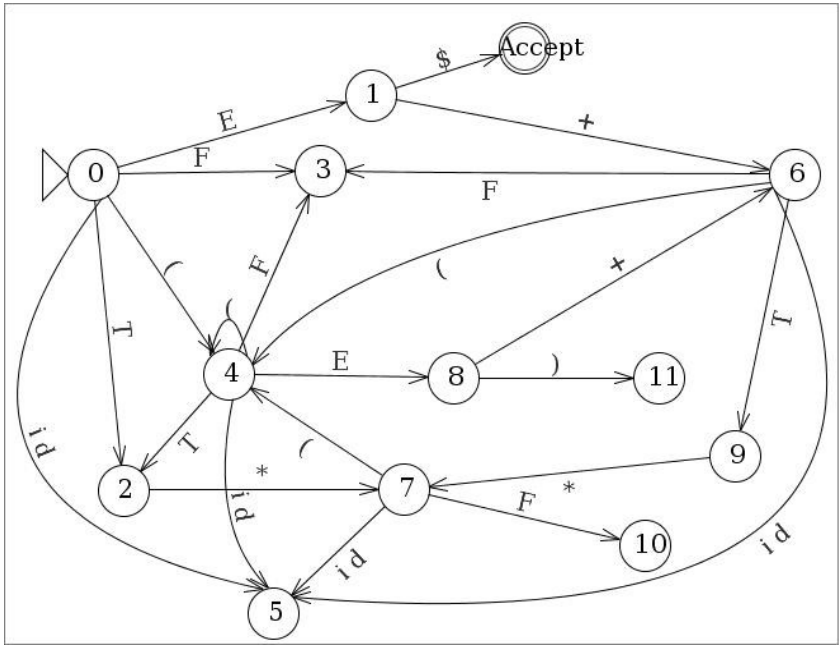



(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I4={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- Transition[I4,id]=I5 donc A[4,id]=d5

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | | | | | | |

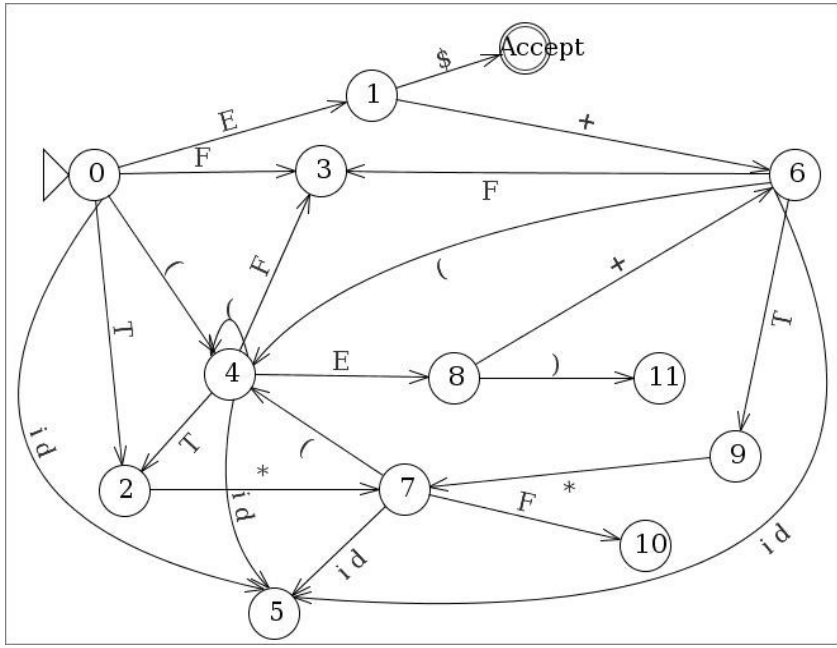



(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I4={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- Transition[I4,id]=I5 donc A[4,id]=d5
- Transition[I4,(]=I4 donc A[4,(]=d4

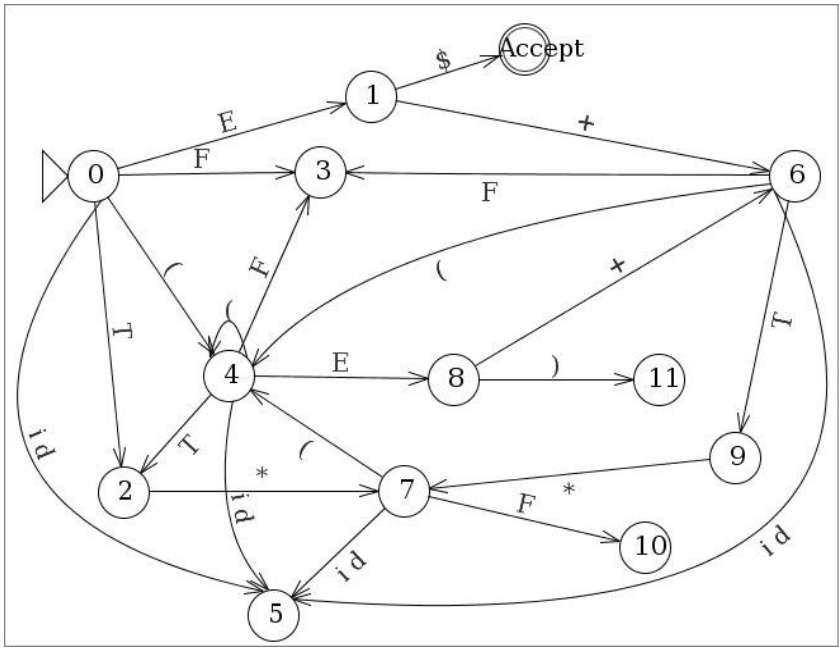
| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | | | |




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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I4={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- Transition[I4,id]=I5 donc A[4,id]=d5
- Transition[I4,(]=I4 donc A[4,(]=d4
- Transition[I4,E]=I8 Successeur[4,E]=8

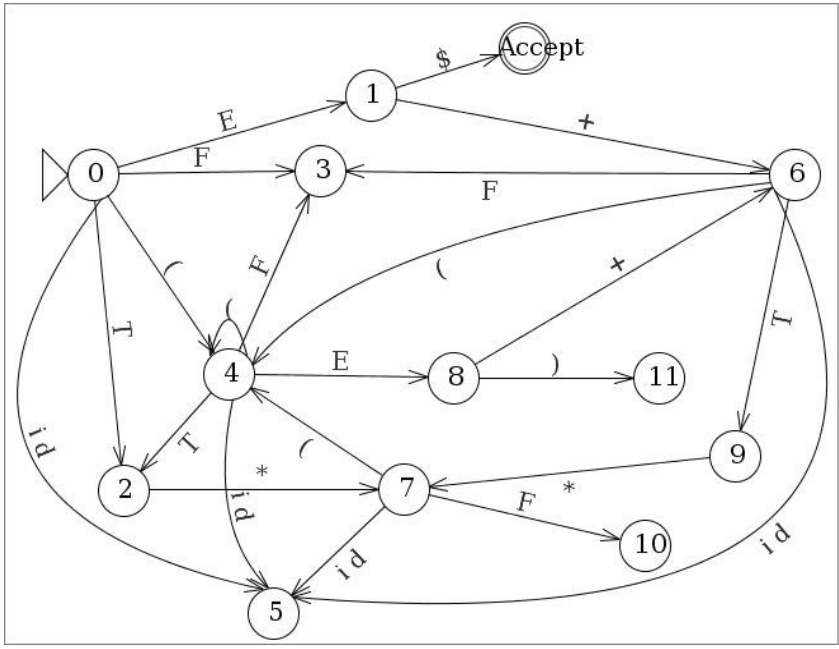



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I4={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- Transition[I4,id]=I5 donc A[4,id]=d5
- Transition[I4,(]=I4 donc A[4,(]=d4
- Transition[I4,E]=I8 Successeur[4,E]=8
- Transition[I4,T]=I2 Successeur[4,T]=2

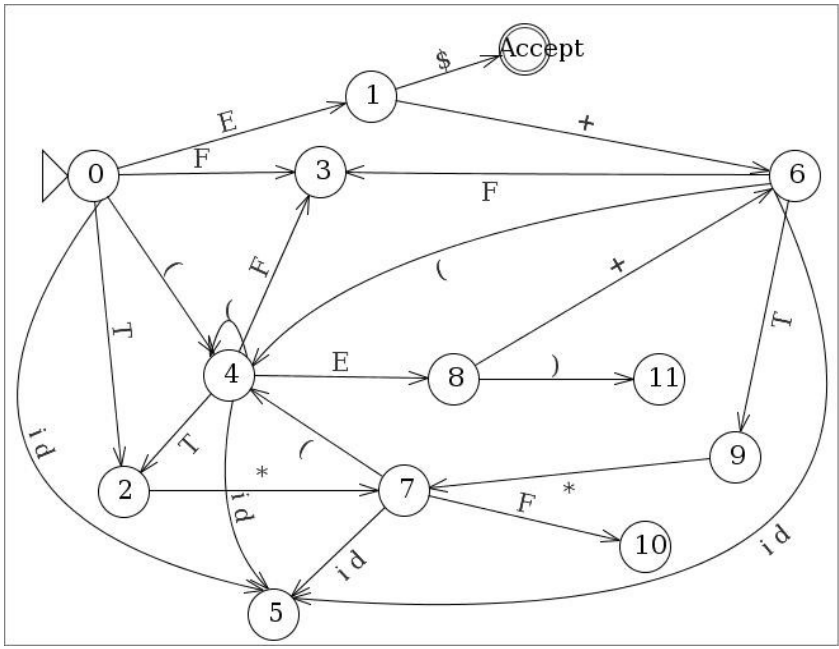



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I4={[F → (·E)], [E → ·E+T], [E → ·T], [T →·T * F], [T → ·F], [F → ·(E)], [F → ·id]}
- Transition[I4,id]=I5 donc A[4,id]=d5
- Transition[I4,(]=I4 donc A[4,(]=d4
- Transition[I4,E]=I8 Successeur[4,E]=8
- Transition[I4,T]=I2 Successeur[4,T]=2
- Transition[I4,F]=I3 Successeur[4,F]=3

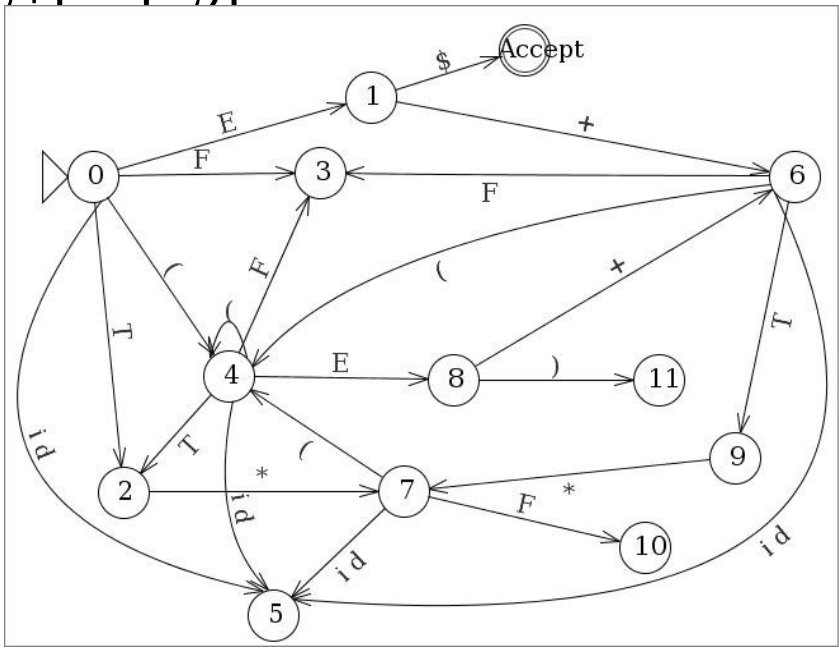



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I5={[F → id·]}
- Suivant(F)={+,*,$,)} donc A[5,+]=A[5,*]=A[5,$]=A[5,)]=r6

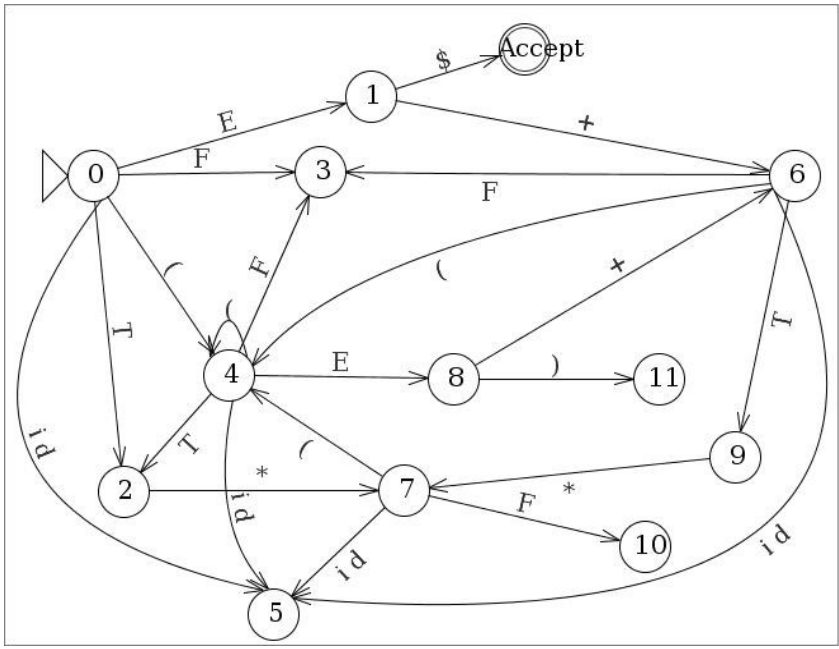



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I6={[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]}
- Transition[I6,id]=I5 alors A[6,id]=d5

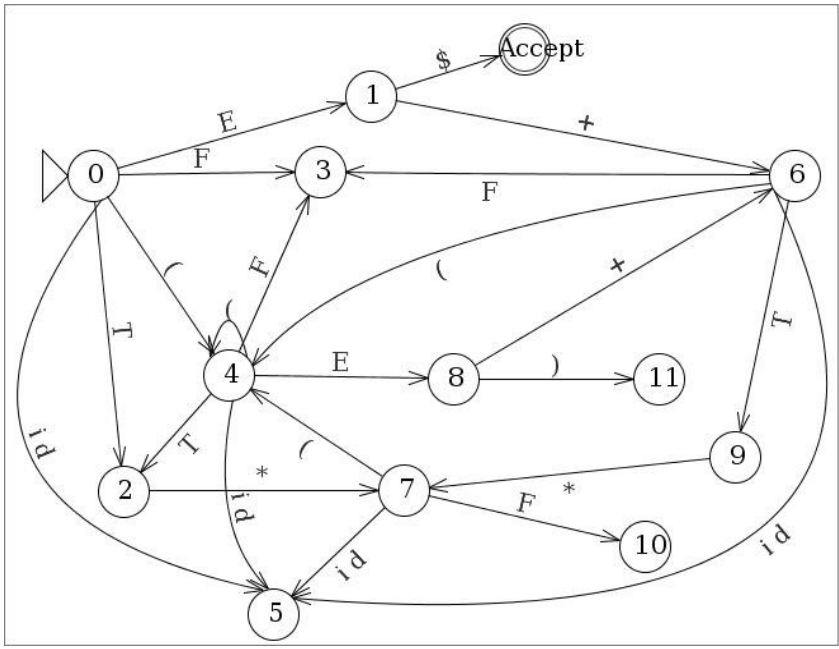



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I6={[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]}
- Transition[I6,id]=I5 alors A[6,id]=d5
- Transition[I6,(]=I4 alors A[6,(]=d4

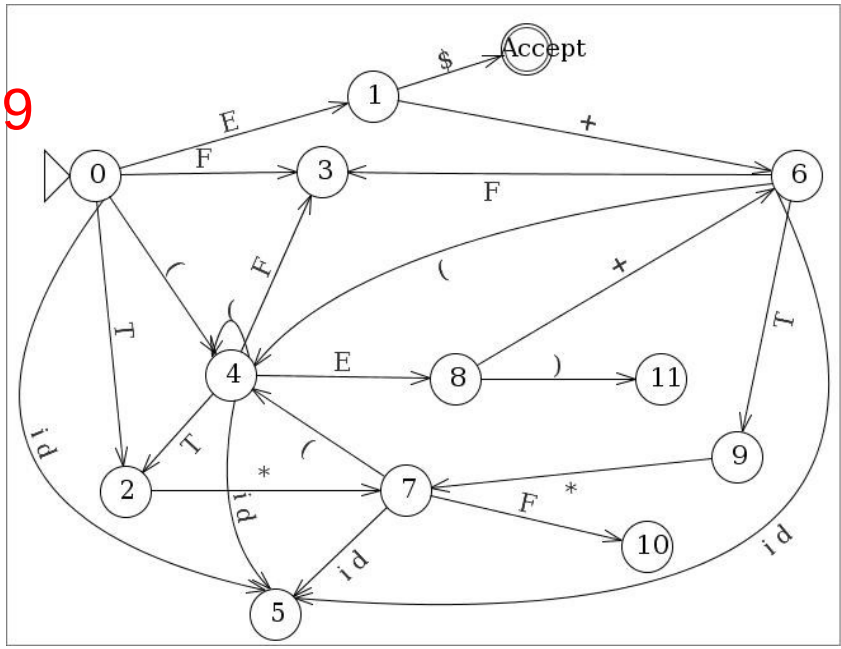



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | d4 | | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I6={[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]}
- Transition[I6,id]=I5 alors A[6,id]=d5
- Transition[I6,(]=I4 alors A[6,(]=d4
- Transition[I6,T]=T9 alors Successeur[6,T]=9

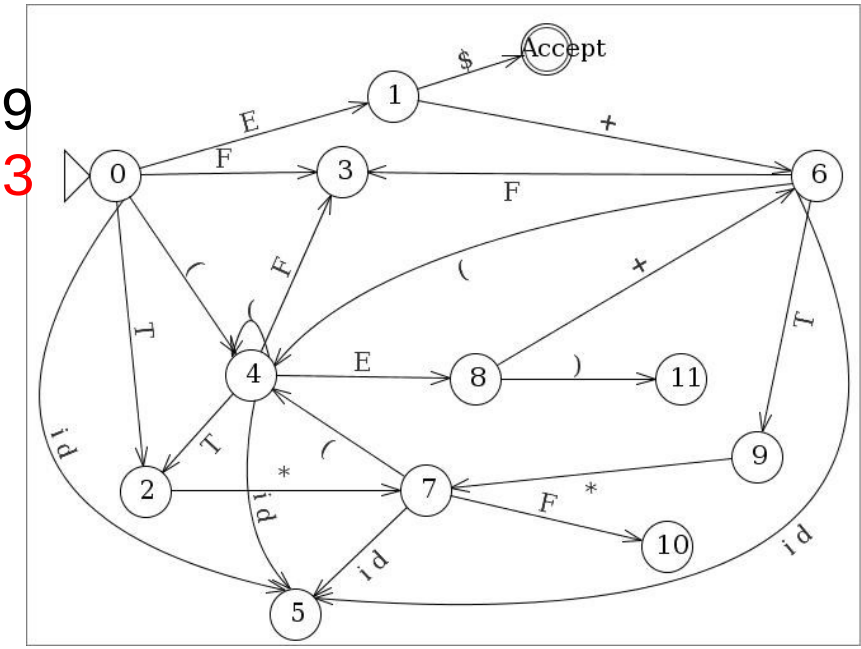



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | d4 | | | | 9 | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I6={[E → E+·T], [T → ·T * F], [T → ·F], [F → ·(E )], [F → ·id]}
- Transition[I6,id]=I5 alors A[6,id]=d5
- Transition[I6,(]=I4 alors A[6,(]=d4
- Transition[I6,T]=I9 alors Successeur[6,T]=9
- Transition[I6,F]=I3 alors Successeur[6,F]=3

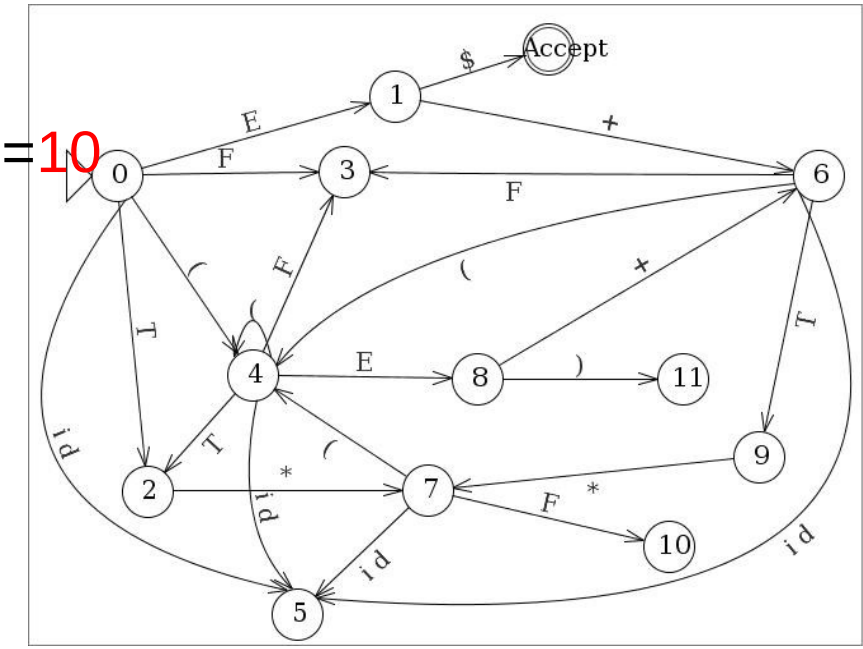



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | d4 | | | | 9 | 3 |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I7={[T → T*·F], [F → ·(E)], [F → ·id]}
- Transition[I7,id]=I5 alors A[7,id]=d5
- Transition[I7,(]=I4 alors A[7,(]=d4
- Transition[I7,F]=I10 alors Successeur[7,F]=10

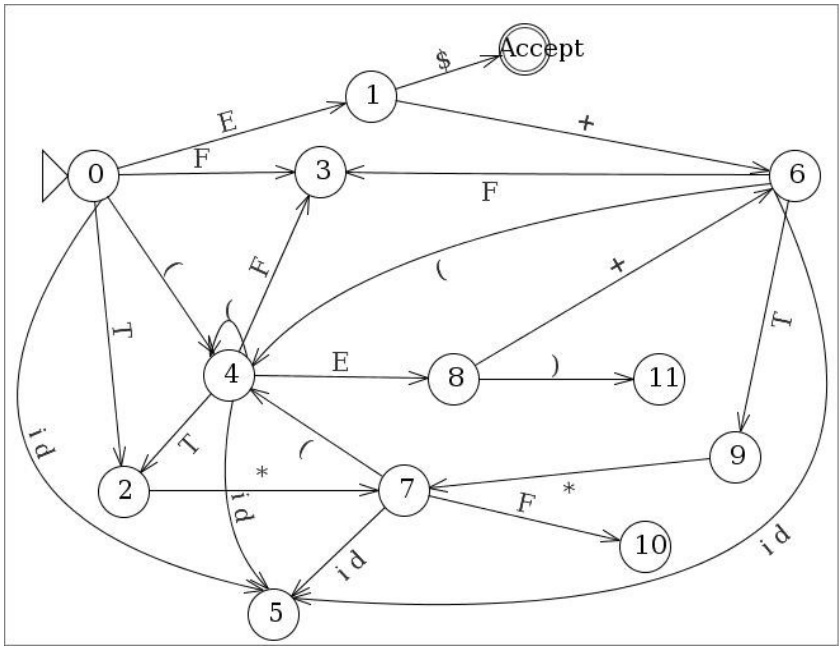



| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | d4 | | | | 9 | 3 |
| 7 | d5 | | | d4 | | | | | 10 |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I8={[F → (E·)], [E → E·+T]}
- Transition[I8,+]=I6 alors A[8,+]=d6
- Transition[I8,)]=I11 alors A[8,)]=d11



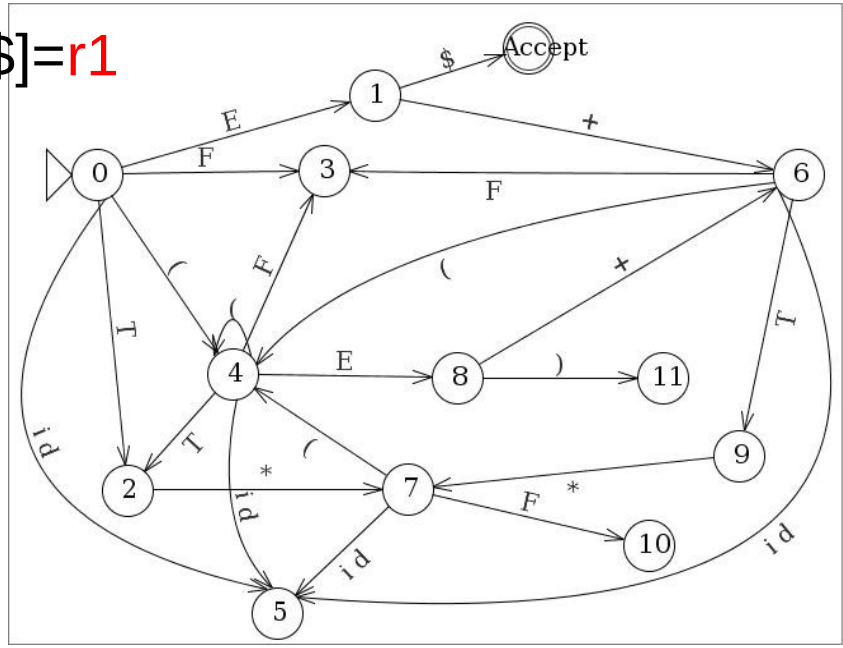

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | E | T | F |
| 0 | d5 | | | d4 | | | 1 | 2 | 3 |
| 1 | | d6 | | | | acc | | | |
| 2 | | r2 | d7 | | r2 | r2 | | | |
| 3 | | r4 | r4 | | r4 | r4 | | | |
| 4 | d5 | | | d4 | | | 8 | 2 | 3 |
| 5 | | r6 | r6 | | r6 | r6 | | | |
| 6 | d5 | | | d4 | | | | 9 | 3 |
| 7 | d5 | | | d4 | | | | | 10 |
| 8 | | d6 | | | d11 | | | | |

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

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I9={[E → E+T·], [T → T· * F]}
- Transition[I9,*]=I7 alors A[9,*]=d7
- Suivant(E)={+,),$} alors A[9,+]=A[9,)]=A[9,$]=r1

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |

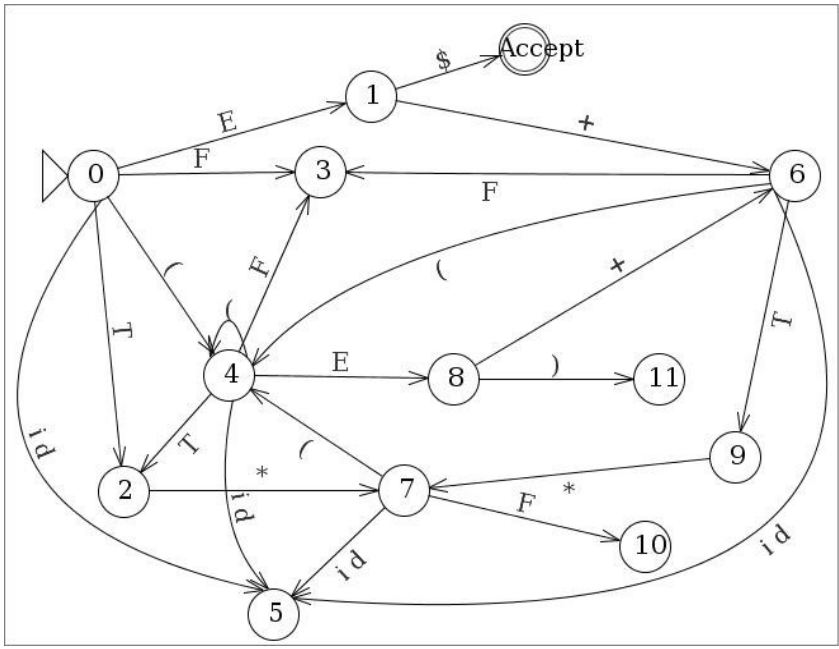



(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I10={[T → T*F·]}
- Suivant(T)={*,+,),$} alors A[10,*]=A[10,+]=A[10,)]=A[10,$]=r3

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |

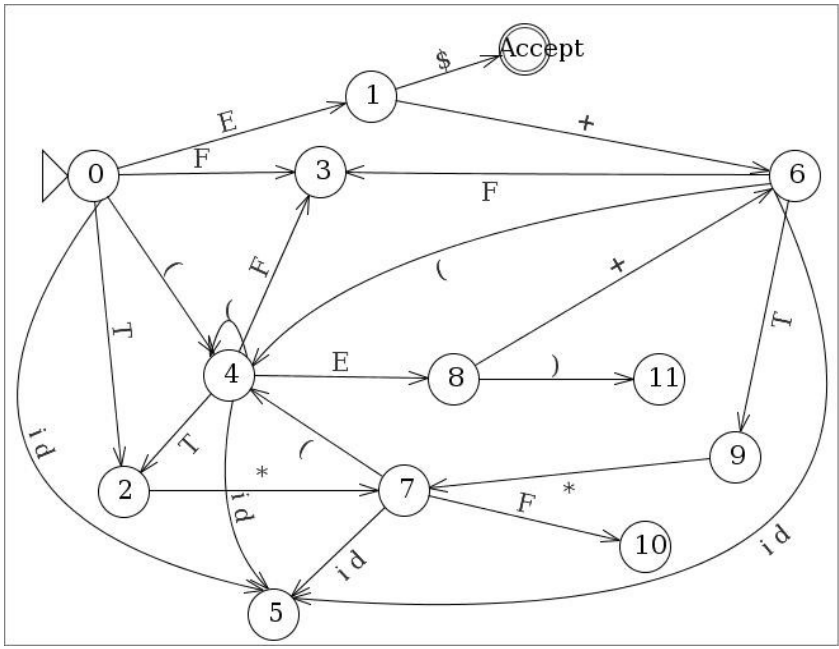



(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I11={[F → (E)·]}
- Suivant(F)={*,+,),$} alors A[11,*]=A[11,+]=A[11,)]=A[11,$]=r5

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |

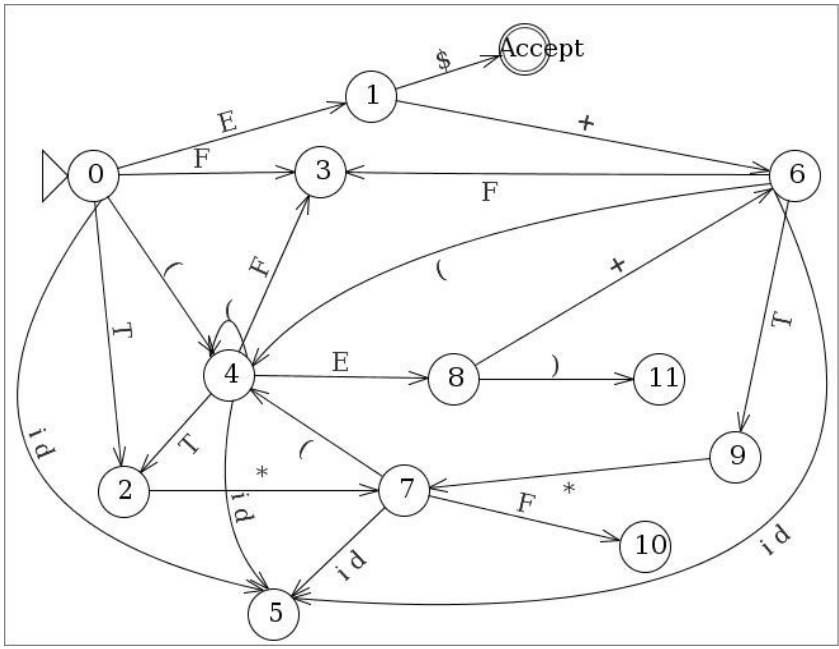



(1) E → E + T
(2) E → T
(3) T → T * F
(4) T → F
(5) F → (E)
(6) F → id

# Construction de la table d'analyse SLR

- **Exemple :**soit la grammaire G' définit précédemment
- I11={[F → (E)·]}
- Suivant(F)={*,+,),$} alors A[11,*]=A[11,+]=A[11,)]=A[11,$]=r5

| État | Action | | | | | | Successeur | | |
|---|---|---|---|---|---|---|---|---|---|
| | id | + | * | ( | ) | $ | 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 | | | |




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

# Limites des grammaires SLR

- **Remarque :** chaque grammaire SLR(1) est une grammaire non ambiguë, mais il existe des grammaires non ambiguë qui ne sont pas SLR(1). Soit la grammaire G définit par :

  S → L = R | R

  L → *R | id

  R → L

- qui code l'affectation et l'accès à la valeur d'un pointeur. Les ensembles d'items de cette grammaire sont :

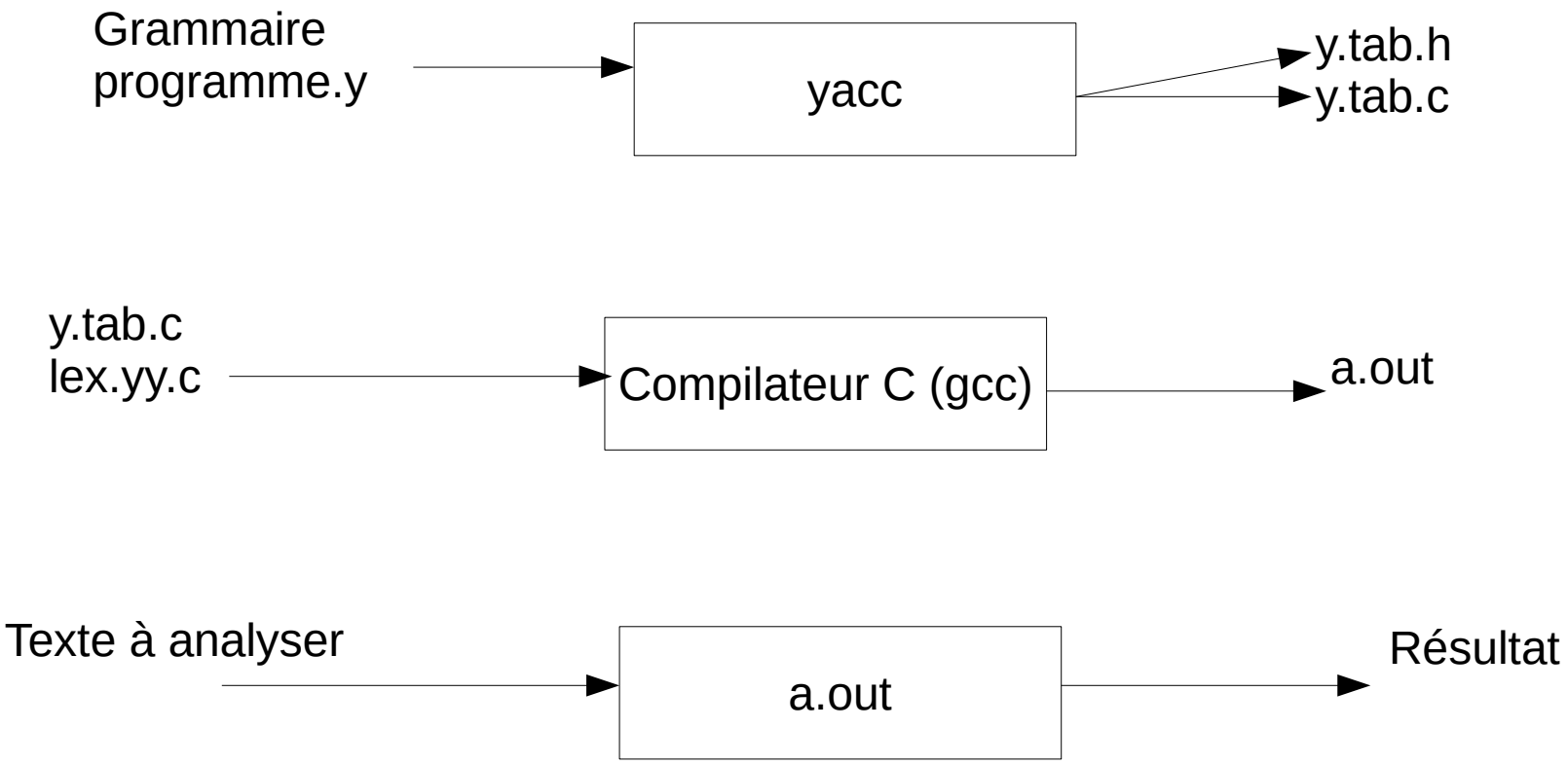
- I0={[S' → ·S],[S → ·L = R],[S → ·R], [L → ·*R],[L → ·id],[R → ·L]}, I1={[S' → S·]}, I2={[S → L· = R],[R → L·]}, I3={[S → R·]}, I4={[L → *·R],[R → ·L],[L → ·*R], [L → ·id]}, I5={[L → id·]}, I6={[S → L = ·R],[R → ·L], [L → ·*R],[L → ·id]}, I7={[L → *R·]}, I8={[R → L·]}, I9={[S → L = R·]}

- L'ensemble I2 d'items de cette grammaire contient : I2={[S → L· = R],[R → L·]} . l'item S → L· = R donne A[2,=]=d6, alors que Suivant(R) contient =, la deuxième action est alors A[2,=]=réduire par R → L, la grammaire G n'est donc SLR(1) et est non ambiguë => conflit de réduire/décaler sur =. la méthode SLR est incapable de décider quelle action prendre sur =

# Limites des grammaires SLR

- Les méthodes canoniques LR(1) et LALR, vont réussir sur une plus vaste collection de grammaires, y compris la grammaire précédente.
- Dans la pratique, les langages de programmation ont des grammaires LALR(1).
- Yacc et Cup construisent des tables pour une grammaire LALR(1).

# Générateur d'analyseur syntaxique Yacc



# Générateur d'analyseur syntaxique Yacc

- Yacc signifie : Yet another compiler-compiler, la première version de yacc date des années 1970 créée par S.C. Johnson. Yacc est disponible comme une commande du système UNIX, et est utilisée pour aider dans l'implémentation des compilateurs.
- Un programme yacc doit être contenu dans un fichier ayant l'extension « .y ». Il a la même structure en trois parties comme lex. Une première partie de déclarations, une seconde partie contient les règles de traduction de l'analyseur syntaxique, et une troisième partie qui contient du code écrit en C qui sera copié dans le programme qui sera généré.

```
%{ déclarations C
%}
Déclarations Yacc
%%
règles de traduction
%%
routines écrites en C
```

# Générateur d'analyseur syntaxique Yacc

- **Partie des déclarations** : Elle contient deux sections optionnelles.
  - La première contient des déclarations ordinaires en C (variables bibliothèques, unions, …) délimitées par %{ et %}.
  - Une seconde section qui contient les déclarations de tokens et d'associativité/priorité des opérateurs.
    - L'axiome avec %start
    - Les tokens de la grammaire avec %token
    - Les opérateurs en précisant leur associativité par %left, %right,%nonssoc
- **Remarques** :
  - Le symbole % doit être à la première colonne
  - Les symboles de la grammaire qui ne sont pas déclarés dans la section de déclarations sont considérés comme des non-terminaux.

# Générateur d'analyseur syntaxique Yacc

- **Partie des règles de traduction** : Cette partie vient après la partie déclarations (optionnelle) après les premiers %%, cette partie contient les règles de production.
- Chaque règle consiste en une production de la grammaire associée avec une action sémantique.
- Un ensemble de production écrites de la façon suivante :

$$\text{<tête>} \rightarrow \text{<corps>}_1 \mid \text{<corps>}_2 \mid \ldots \mid \text{<corps>}_n$$

- peuvent être écrite en Yacc comme suit :

```
<tête> :  <corps>1 {<action sémantique>1}
       |  <corps>2 {<action sémantique>2}
       | …
       | <corps>n {<action sémantique>n}
       ;
```

# Générateur d'analyseur syntaxique Yacc

- **Partie des règles de traduction :**
- Dans une production Yacc, une chaîne de caractères ou de chiffres non déclarés entre quottes ("") comme des tokens sont considérés comme des non-terminaux.
- Un caractère entre simples quottes 'c' est considéré comme un symbole terminal c. Les corps alternatifs sont séparés par une barre verticale (|) et un point virgule (;) suit chaque tête avec ses dérivations et ses actions sémantiques.
- Le premier token est considéré comme le token de départ (axiome).
- Une action sémantique Yacc est une séquence d'instructions écrites en langage C.
- Dans une action sémantique le symbole $$ fait référence à la valeur d'attribut associé au non-terminal de la tête (gauche), tandis que $i fait référence à la valeur associée avec le $i^{ème}$ symbole de la grammaire (terminal ou non-terminal) du corps.

# Générateur d'analyseur syntaxique Yacc

- **Partie routines écrites en C** : :
  - Cette partie contient des routines écrites en C.
  - Elle doit contenir la fonction d'analyse lexicale yylex() dans le cas où Lex est utilisé pour produire l'analyseur lexical, il suffit d'inclure le fichier lex.yy.c par :

```
#include ''lex.yy.c''
```

  - Dans cette partie, des procédures de traitement d'erreur peuvent être introduites si nécessaire.
  - Elle contient la fonction main(), qui ne doit pas être introduite dans le fichier Flex.

# Générateur d'analyseur syntaxique Yacc

- **Exemple** : Considérons la grammaire des expressions arithmétiques suivante :

  S → E fin

  E → E+T | T

  T → T * F | F

  F → (E) | nbre

# Générateur d'analyseur syntaxique Yacc

- **Exemple** : Partie déclarations

parser.y

```
%{
#include<stdio.h>
#include<stdlib.h>
void yyerror(char *);
int yylex();
%}
%union {double dval; int ival;}
%token <dval> nbre
%token fin lparn rparn bye
%type <dval> E T F
%left plus moins
%left fois divs
%start S
```

Les tokens de la grammaire

Les non terminaux de la grammaire

+ et – sont associative à gauche et ont la priorité la plus basse

* et / sont associative à gauche et ont la priorité la plus haute

S est l'axiome sinon le premier symbole sera désigné comme axiome de la grammaire

# Générateur d'analyseur syntaxique Yacc

- **Exemple :** règles de traduction

```
%%
S: E fin {printf("=%f\n",$1);return 0;/*fin */}
;
E: E plus T {$$ = $1 + $3;}
| E moins T {$$ = $1 - $3;}
| T {$$ = $1;}
;
T: T fois F {$$ = $1 * $3;}
| T divs F {if($3 !=0) $$ = $1 / $3; else yyerror("division par 0\n");}
| F {$$ = $1;}
;
F: lparn E rparn {$$ = $2;}
| nbre {$$ = $1;}
| bye {exit(0);}
;
```

# Générateur d'analyseur syntaxique Yacc

- **Exemple :** Routines en C

```
%%

void yyerror(char *msg){
        printf(">> %s\n",msg);
}

int main(int argc,char *argv[])
{       extern FILE *yyin;
        if(argc>1){
                FILE *file;
                file = fopen(argv[1],"r");
                if(!file){
                fprintf(stderr,"fichier ne peut être ouvert%s\n",argv[1]);
                exit(1);
                }
        yyin=file;
        }
        else yyin=stdin;
        while(1){
                yyparse();
        }
        return 0;
}
```

# Générateur d'analyseur syntaxique Yacc

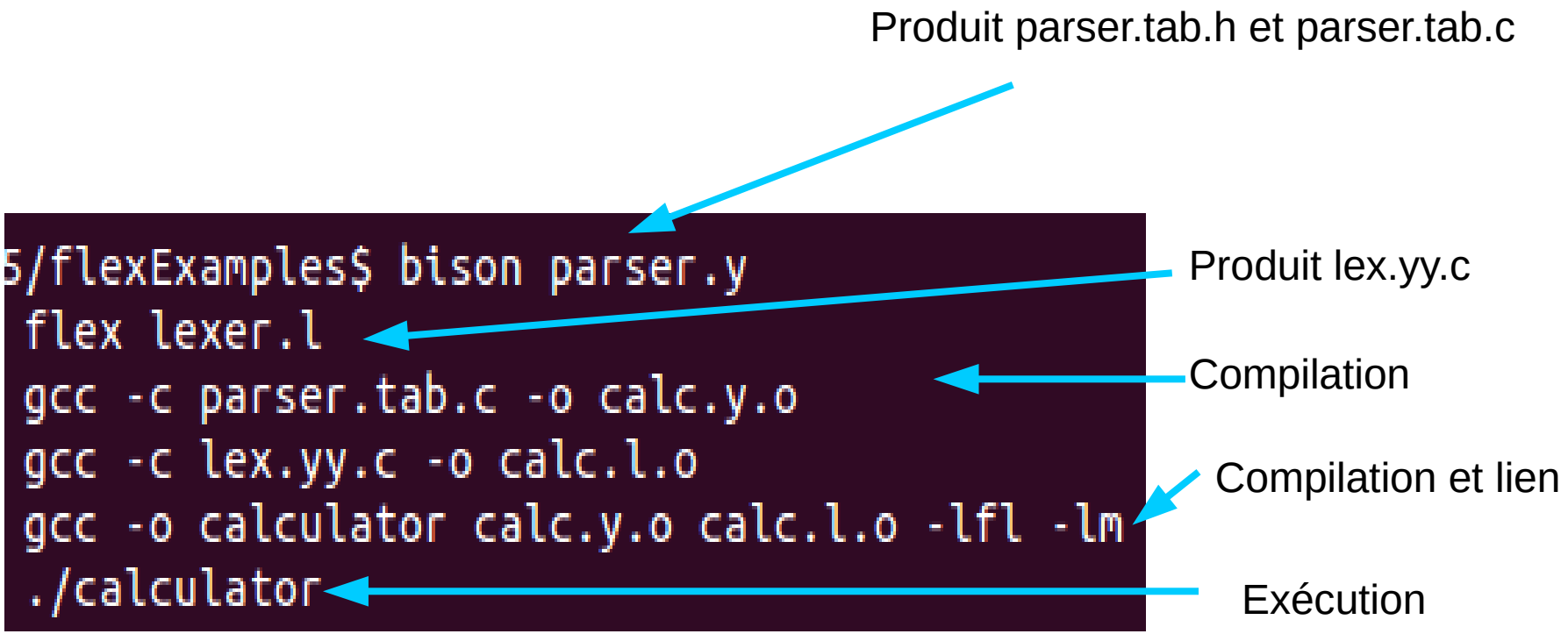
- **Exemple :** L'analyseur lexical

lexer.l

```
/*%option noyywrap /*Utiliser sours Windwos*/
%{
#include<stdlib.h>
#include "parser.tab.h"
int yywrap(void) {} /*Utiliser sous GNU Linux*/
%}
nbre [0-9]+|[0-9]*\.[0-9]+
%%
[ \t]+ {/*ne rien faire*/}
{nbre} {yylval.dval = atof(yytext); return nbre;}
"\n" {return fin;}
"+" {return plus;}
"-" {return moins;}
"/" {return divs;}
"*" {return fois;}
"(" {return lparn;}
")" {return rparn;}
"bye" {return bye;}
```

# Générateur d'analyseur syntaxique Yacc

- **Exemple :** Exécution



# Générateur d'analyseur syntaxique Yacc

- **Exemple :** Exécution

```
1-6 + (6-8) * 9 - (2/9) + 12
=-11.222222
3+6
=9.000000
9*8
=72.000000
98/7
=14.000000
12-9
=3.000000
(3+8) - (8+9) - (5 / 9) + 6
=-0.555556
bye
```

# Analyse sémantique

- Un constituant important de l'analyse sémantique est le contrôle de type, un contrôleur de type vérifie que le type d'une construction correspond bien au type attendu par son contexte.
- Par exemple l'opérateur arithmétique modulo nécessite des opérandes entières, le contrôleur de type doit vérifier que les opérandes de modulo sont de type entier.
- De façon similaire le contrôleur de type doit vérifier que toute fonction définit par l'utilisateur est utilisée avec le bon nombre d'arguments et de type correcte.

# Analyse sémantique

- Dans la plupart des langages les types peuvent être soit des types de base soit des types construits :
  - les types de base sont les types atomiques sans structure interne ou dont la structure interne n'est pas accessible aux programmeurs. Par exemple entier, réel, …
  - les types construits à partir des types de base ou d'autres types construits comme les tableaux, les structures, les fonctions, …
- Le type de toute construction d'un langage est dénotée par une expression de type.
- On introduit dans les contrôleurs de type un symbole de typage qui est une collection de règles permettant d'associer des expressions de type aux diverses parties du programme.

# Analyse sémantique

- **Exemple :**

  P → D ; E

  D → D ; D | id : T

  T → caractères | entier | tableau [nb] de T

  E → nb | id | E mod E | E [E]

  var T : array [1..4] of integer ;