Contents

[Fichier AD.cpp 1](#_Toc196171741)

[Les Structure de données 1](#_Toc196171742)

[Les Fonctions : 1](#_Toc196171743)

[1) Initialisation de FOLLOW 1](#_Toc196171744)

[2) Boucle principale : propagation jusqu’à stabilisation 1](#_Toc196171745)

[3) Parcours de toutes les productions 1](#_Toc196171746)

[4) Quand on trouve un non-terminal dans une production 1](#_Toc196171747)

[a) Cas 1 : Il est suivi d’un terminal 1](#_Toc196171748)

[b) Cas 2 : Il est suivi d’un autre non-terminal 1](#_Toc196171749)

[c) Cas 3 : Le non-terminal est à la fin de la production 1](#_Toc196171750)

[Execution et Testing 2](#_Toc196171751)

[Implémentation de Analyseur Syntaxique Descendante 3](#_Toc196171752)

[Test et Explication de code AA1.cpp 4](#_Toc196171753)

[Objectives 4](#_Toc196171754)

[Définitions et structures de données 4](#_Toc196171755)

[Règle de Grammaire 5](#_Toc196171756)

[LR1 State 6](#_Toc196171757)

[Table LR(1) 6](#_Toc196171758)

[Grammaire définie 6](#_Toc196171759)

[Construction des états LR(1) 7](#_Toc196171760)

[Construction de la table LR(1) 8](#_Toc196171761)

[Première Exécution de code AA1.cpp 9](#_Toc196171762)

[Complétion de codes 10](#_Toc196171763)

[Fonction Closure 10](#_Toc196171764)

[Fonction goto\_state 10](#_Toc196171765)

[Deuxième Exécution de code AA1.cpp 11](#_Toc196171766)

[Implémentation de la Grammaire de l’exercice 4 13](#_Toc196171767)

[Test et Explication de code AA2.cpp 14](#_Toc196171768)

[Définitions et structures de données 14](#_Toc196171769)

[Limite de Parseur AA2.cpp 16](#_Toc196171770)

[Implementation et Execution de grammaire de l’exercice 4 sur le Code AA2.cpp 17](#_Toc196171771)

[Conclusion : 17](#_Toc196171772)

[Figure 1: Affichage de l’Execution de la fichier AD.cpp par le grammaire M. 5](#_Toc196171773)

[Figure 2: Affichage de l’exécution de la fonction readSyntaxe 6](#_Toc196171774)

[Figure 3: Structure Règle de grammaire 7](#_Toc196171775)

[Figure 4: Structure LRItem 7](#_Toc196171776)

[Figure 5: Structure LR1State 8](#_Toc196171777)

[Figure 6: Structure LR1Table 8](#_Toc196171778)

[Figure 7: Grammaire de départ 9](#_Toc196171779)

[Figure 8: Build\_lr1\_States Function 9](#_Toc196171780)

[Figure 9: Build\_lr1\_table Function 10](#_Toc196171781)

[Figure 10: AA1.cpp fonction main 11](#_Toc196171782)

[Figure 11:Premiere Affichage 11](#_Toc196171783)

[Figure 12: fonction main a exécuter 14](#_Toc196171784)

[Figure 13: Deuxième affichage du fichier AA1.cpp 15](#_Toc196171785)

[Figure 14: Grammaire Exercice 4 TD2.2 15](#_Toc196171786)

[Figure 15: Table LR(1) de la Grammaire d’exercice 4 16](#_Toc196171787)

[Figure 16: Symboles de Grammaire 16](#_Toc196171788)

[Figure 17: Les Action de Table LR(1) 17](#_Toc196171789)

[Figure 18: Structure Action 17](#_Toc196171790)

[Figure 19: Structure d’une regle 17](#_Toc196171791)

[Figure 20:Grammaire a adoptee 17](#_Toc196171792)

[Figure 21: Affichage de l’analyseur ascendante de l’exercice 4 19](#_Toc196171793)

[Tableau 1:Tableau LR(1) 18](#_Toc196171794)

# Fichier AD.cpp

Les Structure de données

Rules: présente les règles de grammaire par un symbole non-terminale.

Grammar : présente le grammaire de la langage.

FirstFollow : présente le précèdent et le suivant d’un symbole non-terminal

LL1TableEntry : présente la structure de la table LL(1) associée a la grammaire concerne.

## Les Fonctions :

calculate\_first

La fonction calculate\_first sert à **calculer les ensembles FIRST** pour chaque non-terminal dans une grammaire.

1. **Initialisation des ensembles FIRST**
2. Boucle principale (itérative jusqu'à stabilité)
3. Traitement de chaque règle de production
   1. **Cas 1 : Le premier symbole est un terminal**
   2. Cas2 : Le premier symbole est un non-terminal

calculate\_follow

Cette fonction calcule l’**ensemble FOLLOW** pour chaque non-terminal d’une grammaire.

1. Initialisation de FOLLOW
2. Boucle principale : propagation jusqu’à stabilisation
3. **Parcours de toutes les productions**
4. Quand on trouve un non-terminal dans une production
   1. Cas 1 : Il est suivi d’un **terminal**
   2. Cas 2 : Il est suivi d’un **autre non-terminal**
   3. Cas 3 : Le non-terminal est à la fin de la production

build\_ll1\_table

La fonction construit la table d’analyse prédictive LL(1) utilisée par un analyseur syntaxique descendant pour déterminer quelle règle appliquer en fonction du non-terminal courant et du symbole d’entrée.

La table LL(1) est remplie selon les règles classiques :

* Si la production commence par un terminal a, on place la production dans la case [NonTerminal][a].
* Si elle commence par un non-terminal A, on place la production dans toutes les cases [NonTerminal][x] où x ∈ FIRST(A).
* Si la production est vide (epsilon), on la place dans toutes les cases [NonTerminal][x] où x ∈ FOLLOW(NonTerminal).

1. **Initialisation du nombre d’entrées**
2. **Parcours de chaque règle de la grammaire**
   1. Cas 1 : La production commence par un **terminal**
   2. Cas 2 : La production commence par un **non-terminal**
   3. Cas 3 : La production est vide (epsilon)

Execution et Testing  
Soit le grammaire (M) suivant :

S → aAS | bB

A → cC | dD

B → b | ε

C → c | ε

D → d | ε

La Figure présente l’affichage de l’exécution de la fichier AD.cpp en appliquant le grammaire ci-dessus.

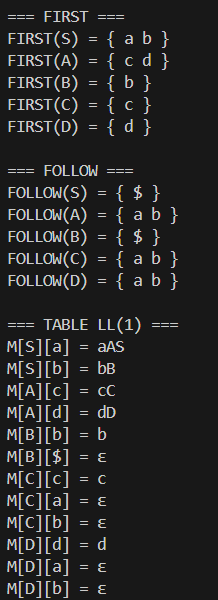


Figure 1: Affichage de l’Execution de la fichier AD.cpp par le grammaire M.

## Implémentation de Analyseur Syntaxique Descendante

LafonctionReadSyntaxe **Vérifier si une chaîne d'entrée est syntaxiquement correcte** selonle **grammaire LL(1) précèdent.**

Étapes de fonctionnement

1. **Initialisation**
   1. Créer une **pile** pour l’analyse syntaxique.
   2. Mettre **le symbole de fin $** tout en bas de la pile.
   3. Mettre ensuite le **symbole de départ** de la grammaire (ex : S) au sommet.
2. **Boucle d’analyse principale**
   1. **Lire le sommet de la pile** et le **symbole courant de l’entrée** (lookahead).
   2. **Cas 1 :** Si top\_stack == current\_input
   3. **Cas 2 :** Si top\_stack est un **non-terminal**
   4. **Cas 3 :** Si top\_stack est un terminal ≠ current\_input
3. Fin de la lecture

Le figure 2 présente l’affichage de l’exécution de la fonction ReadSyntaxe qui accepte seulement les mots qui appartient au langage associe a la grammaire (M).

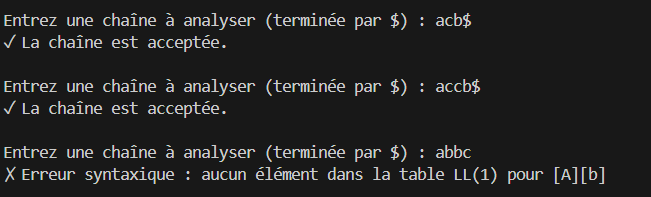


Figure 2: Affichage de l’exécution de la fonction readSyntaxe

# Test et Explication de code AA1.cpp

## Objectives

Tester et expliquer le code AA1.cpp

Compléter et appliquer ces codes pour une analyse ascendante des grammaires présentées dans

les exercices 3 et 4 du TD2.2.

vérifier si ce code peut effectuer une analyse syntaxique ascendante à n’importe

qu’elle grammaire.

### Définitions et structures de données

Ce code implémente la construction d’un analyseur syntaxique LR(1) pour une grammaire donnée. Il génère d’abord l’ensemble des états LR(1) en construisant des éléments augmentés avec leurs symboles de prévision (lookaheads). Ensuite, il établit automatiquement les tables d’analyse syntaxique : la table action\_table pour les transitions sur les symboles terminaux (shift, reduce, accept) et la table goto\_table pour les transitions sur les symboles non-terminaux. Ces structures sont essentielles pour permettre l’analyse correcte d’une expression selon la grammaire définie.

### Règle de Grammaire

La figure 3 respresente la definition de la regle de la grammaire.

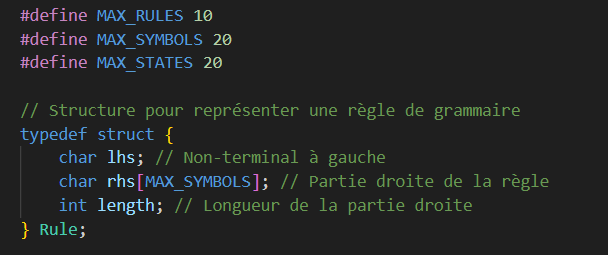


Figure 3: Structure Règle de grammaire

Prenant l’exemple de E -> E + T Alors on aura :

Lh = ‘E’

Rhs = ‘E + T’

Length = 3

#### LR1 Item

La figure 4 présente la définition de la

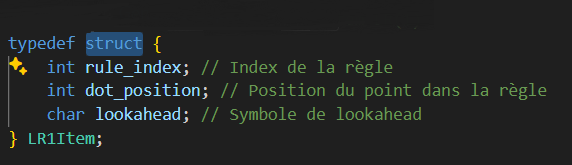


Figure 4: Structure LRItem

**Item LR(1) (LR1Item)** : Représente un point de progression dans une règle, avec un lookahead.

* **rule\_index**: identifie la règle
* **dot\_position**: position du point dans la RHS
* **lookahead**: symbole terminal attendu après

### LR1 State

La figure 5 présente la définition de la l’élément LRState qui définissent un état de l’automate.

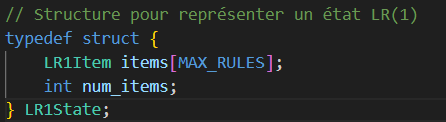


Figure 5: Structure LR1State

### Table LR(1)

La figure 6 présente la définition de l’élément LR1Table.

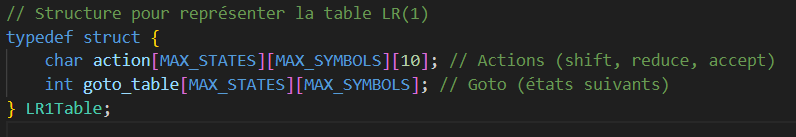


Figure : Structure LR1Table

**action**: table d’actions (shift, reduce, accept)

**goto\_table**: transitions sur non-terminaux (non utilisé ici)

## Grammaire définie

La figure 7 présente le grammaire en définissant les règles de départ.

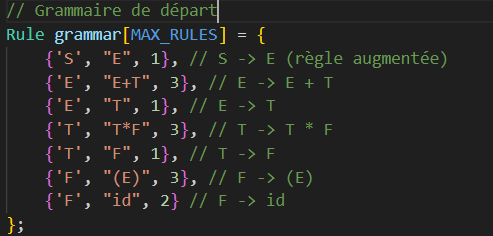


Figure 7: Grammaire de départ

## Construction des états LR(1)

La figure 8 présente la fonction de construction des états LR(1) de la grammaire de figure 7

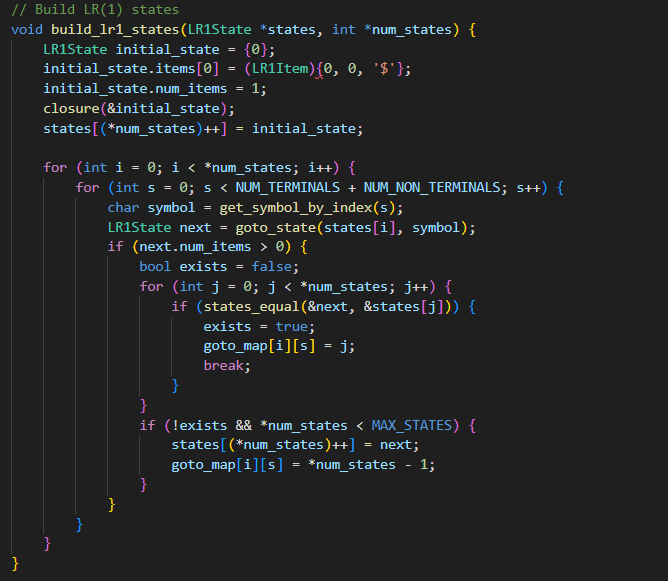


Figure 8: Build\_lr1\_States Function

Crée le premier état contenant l’item de départ avec $ comme lookahead.

Appelle closure() pour compléter l’état.

Ensuite, parcourt les symboles de a à z pour créer des transitions via goto\_state().

## Construction de la table LR(1)

La figure 9 présente la fonction de construction de la table lr1.

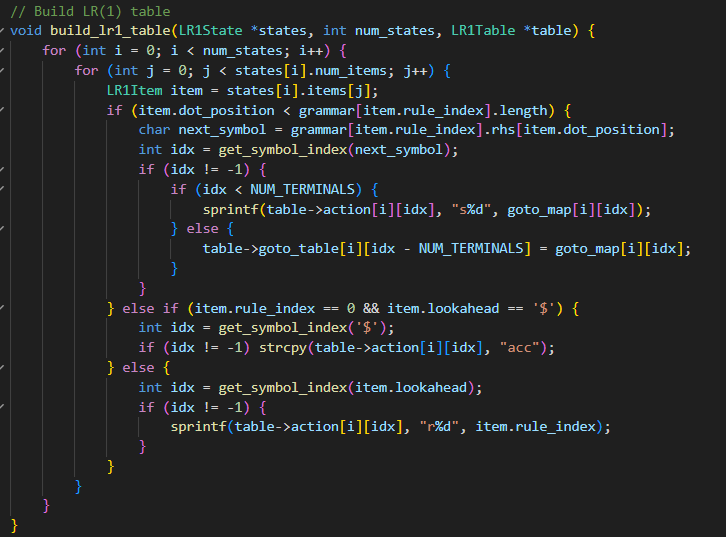


Figure : Build\_lr1\_table Function

* Si le point est au milieu, on effectue un décalage (shift).
* Si c’est la règle initiale terminée, et le lookahead est $, on accepte.
* Sinon, on applique une réduction.

## Première Exécution de code AA1.cpp

La figure 10 présente la fonction main du fichier AA1.cpp qui va être exécuter.

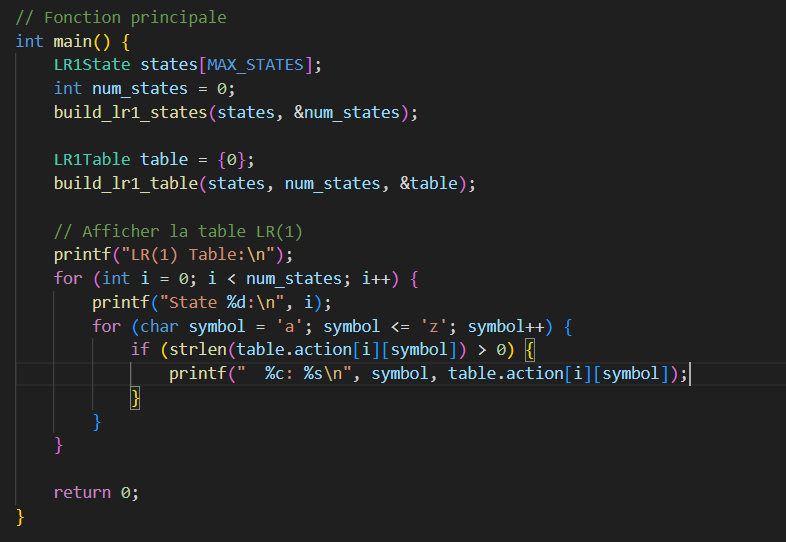


Figure 10: AA1.cpp fonction main

La figure 11 présente l’affichage de la fonction main de figure 10.



Figure :Premiere Affichage

**Remarque** : les fonctions closure() et goto\_state() sont incomplètes (elles ne font rien ou presque), ce qui bloque la progression de la construction de l’automate LR(1).

**Conclusion** : Il faut compléter les fonction closure() et goto\_State() pour une affichage correcte.

## Complétion de codes

Dans cette partie on va procède a la complétion de codes closure et goto\_state().

### Fonction Closure

La **fermeture** d’un ensemble d’items LR(1) consiste à **ajouter récursivement des items** pour tous les non-terminaux qui apparaissent **immédiatement après le point** dans un item existant , les Etapes de code est comme suit :

* Boucle while (added) :  
  On continue tant qu’on ajoute des items.
  + On parcourt chaque item déjà présent dans l’état.
  + On regarde le symbole juste après le point.  
    S’il s’agit d’un **non-terminal** (ex. 'E', 'T', 'F'), on veut explorer ses productions.
  + Pour chaque règle ayant ce non-terminal à gauche, on crée un **nouvel item** avec :
  + le point au début (**dot\_position = 0**)
  + le même lookahead
  + On vérifie s’il est **déjà présent** pour éviter les doublons.
  + Si ce n’est pas le cas, on **ajoute** l’item.

### Fonction goto\_state

À partir d’un état courant I et d’un symbole X, on veut construire un nouvel état :

en déplaçant le point (•) au-delà de X pour chaque item de I où le point précède ce symbole, puis on applique la fermeture sur le résultat pour capturer tous les

items dérivés. Les étapes de code :

* On parcourt tous les items de l’état source.
  + On sélectionne ceux dont le point est juste avant le symbole symbol.
  + Pour chaque item éligible, on crée un nouvel item avec le point avancé d’un cran :
  + Exemple : E → E • + T , $ devient E → E + • T , $ si le symbole est '+'.
  + On ajoute ce nouvel item à l’état cible.
  + Une fois tous les items déplacés, on appelle closure(&new\_state) pour compléter le nouvel état.

## Deuxième Exécution de code AA1.cpp

D’après la complétion de des deux codes closure et goto\_state, on a fait l’exécution de la fichier AA1.cpp après les changement effectuer. On donne la nouvelle fonction main du fichier dans la figure 12.

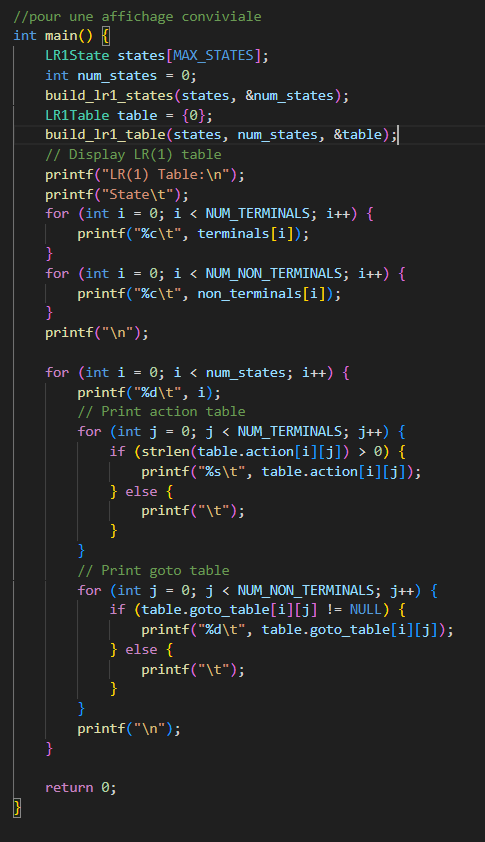


Figure 12: fonction main a exécuter

La figure 13 présente l’affichage de la fonction main exécutée.

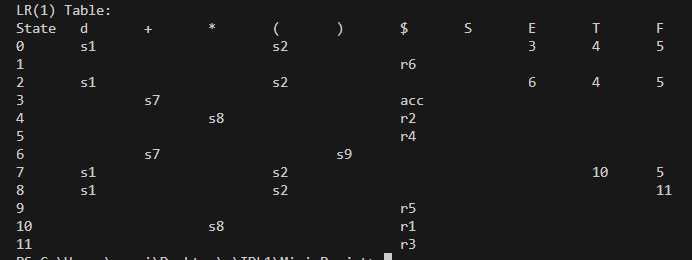


Figure 13: Deuxième affichage du fichier AA1.cpp

Les changement effectuée sont:

* Le traitement des symboles majuscules (non-terminales)
* Le traitement des symboles spéciales (+ \*)
* Le remplacement de symbole d au lieu de ‘id’ (identifiant).
* Rendre l’affichage plus lisible et compréhensible

## Implémentation de la Grammaire de l’exercice 4

La figure 14 présente la grammaire de l’exercice 4 du TD 2.2

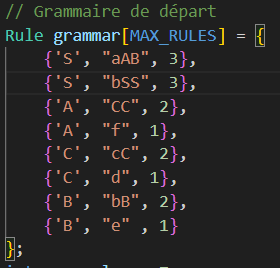


Figure : Grammaire Exercice 4 TD2.2

La figure 15 présente la table LR(1) de la grammaire de figure 14 après l’exécution de fonctions de la fichier AA1.cpp

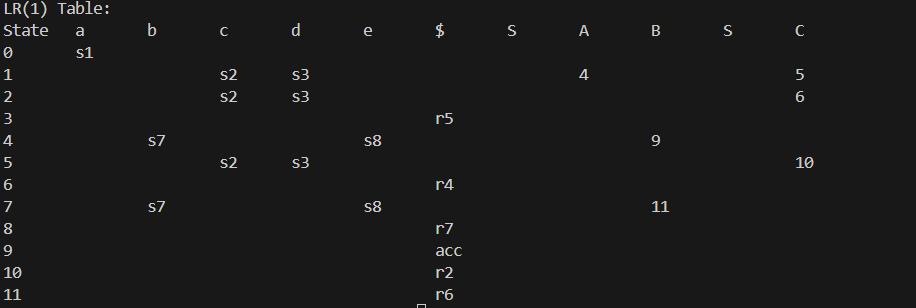


Figure 15: Table LR(1) de la Grammaire d’exercice 4

# Test et Explication de code AA2.cpp

Le code construit la table d’analyse LR(1), composée de action\_table et goto\_table, qui détermine les actions à effectuer lors de l’analyse d’une expression (décalage, réduction ou acceptation). Une fois cette table générée, elle est utilisée pour analyser une entrée et vérifier automatiquement si celle-ci est syntaxiquement correcte selon la grammaire définie. Cela permet d’implémenter un parseur efficace et fiable pour reconnaître les structures du langage.

### Définitions et structures de données

La figure 16 présente les symbole terminaux et non-terminaux de grammaire concerne.

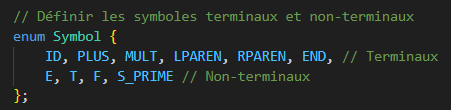


Figure 16: Symboles de Grammaire

La figure 17 présente les action de la table LR(1).

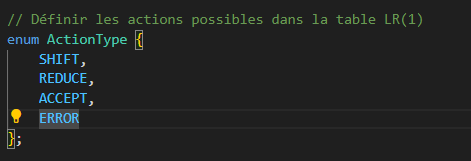


Figure 17: Les Action de Table LR(1)

La figure 18 présente la structure de représentation d’une action au sein du table LR(1).

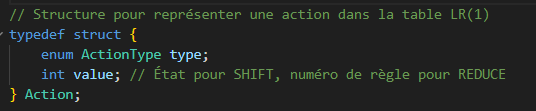


Figure 18: Structure Action

La figure 19 présente la structure de la règle d’un grammaire.

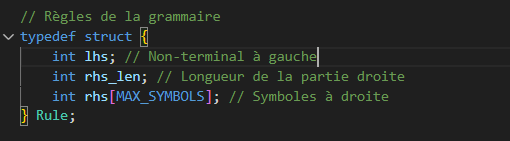


Figure 19: Structure d’une regle

La figure 20 présente la définition de la grammaire a adopte.

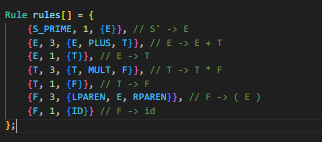


Figure 20:Grammaire a adoptee

La table 1 présente le tableau LR(1) implémente.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **State** | **id (0)** | **+ (1)** | **\* (2)** | **( (3)** | **) (4)** | **$ (5)** | **E (6)** | **T (7)** | **F (8)** |
| 0 | s5 |  |  | s4 |  |  | 1 | 2 | 3 |
| 1 |  | s6 |  |  |  | acc |  |  |  |
| 2 |  | r2 | s7 |  | r2 | r2 |  |  |  |
| 3 |  | r4 | r4 |  | r4 | r4 |  |  |  |
| 4 | s5 |  |  | s4 |  |  | 8 | 2 | 3 |
| 5 |  | r6 | r6 |  | r6 | r6 |  |  |  |
| 6 | s5 |  |  | s4 |  |  |  | 9 | 3 |
| 7 | s5 |  |  | s4 |  |  |  |  | 10 |
| 8 |  | s6 |  |  | s11 |  |  |  |  |
| 9 |  | r1 | s7 |  | r1 | r1 |  |  |  |
| 10 |  | r3 | r3 |  | r3 | r3 |  |  |  |
| 11 |  | r5 | r5 |  | r5 | r5 |  |  |  |

Tableau 1:Tableau LR(1)

### Limite de Parseur AA2.cpp

* Table LR codée en dur :
  + La table de parsing (action\_table et goto\_table) est pré-calculée manuellement pour une grammaire spécifique.
  + Pour une autre grammaire, il faudrait recalculer et réécrire toute la table.
* Grammaire fixe :
  + Les règles de production sont hardcodées dans le tableau rules[].
  + Exemple : E → E + T est écrit en dur, impossible de modifier sans recompiler.
* Symboles prédéfinis :
  + Les terminaux (ID, PLUS, etc.) et non-terminaux (E, T, etc.) sont fixes.
* Pas de génération automatique :
* Un vrai analyseur LR(1) générique construit automatiquement la table à partir des règles de grammaire.

## Implementation et Execution de grammaire de l’exercice 4 sur le Code AA2.cpp

Nous avons procédé à la modification de la définition de la grammaire en adoptant celle fournie dans l’exercice 4 du TD 2.2. De plus, nous avons adapté la table LR(1) en nous basant sur la figure 15.  
La figure 21 présente quelques mots analysés par l’analyseur syntaxique ascendant contenu dans le fichier **AA2EX4.cpp**.

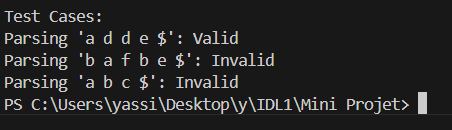


Figure 21: Affichage de l’analyseur ascendante de l’exercice 4

# Conclusion :

Les fichiers **AA1.cpp** et **AA2.cpp** peuvent effectuer une analyse syntaxique ascendante pour n’importe quelle grammaire, à condition que le code soit adapté à la grammaire concernée. Autrement dit, il est nécessaire de modifier le code en fonction de la grammaire utilisée.