**Langages et Compilation**

***Mini projet***

**TORTEL Paul – PILAUDEAU Matthieu – COURSAC Vincent**

L3 – **Équipe F7**

Table des matières

[Fichier d’entrée 1](#_Toc480477006)

[Structures de données 2](#_Toc480477007)

[Algorithmes 4](#_Toc480477008)

[A. Lecture du fichier 4](#_Toc480477009)

[B. Elimination de la récursivité à gauche 5](#_Toc480477010)

[C. Trouver les Premiers des symboles NT 6](#_Toc480477011)

[D. Trouver les Suivants des symboles NT 7](#_Toc480477012)

[E. Créer la table d’analyse 8](#_Toc480477013)

[F. Analyser une phrase 9](#_Toc480477014)

[Traces d’exécutions 10](#_Toc480477015)

**Bilan**

|  |  |
| --- | --- |
| **Lecture de la grammaire initiale** | Fonctionne |
| **Élimination de la récursivité à gauche** | Fonctionne |
| **Calcul des *premiers*** | Fonctionne |
| **Calcul des *suivants*** | Fonctionne |
| **Construction de la table d’analyse** | Fonctionne |
| **Analyse d’une chaîne en entrée** | Fonctionne |

# **Fichier d’entrée**

Pour représenter une grammaire dans notre programme, il faut la décrire dans un fichier texte correspondant en respectant le format que nous avons choisis. Ainsi, nous avons un fichier par grammaire.

Chaque ligne du fichier texte décrit un symbole non terminal. Exemple :

Expreb = Expreb ou Termeb | Termeb

Cette ligne peut se décomposer de la manière suivante :

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Expreb** |  | **=** |  | **Expreb** |  | **ou** |  | **Termeb** |  | **|** |  | **Termeb** |

*Chaque expression ou operateur est séparé par un espace. Cela facilite la lecture du fichier et clarifie la représentation.*

Il y a deux *opérateurs* gérés dans notre format : **=** et **|**

**=** permet de représenter la flèche 🡪, qui a pour but d’indiquer quel est le non terminal décrit par les règles situées à droite de cet opérateur.

**|** permet de représenter l’opérateur logique « OU », qui permet de créer plusieurs règles pour un symbole non terminal. Dans notre exemple, le symbole Expreb possède donc deux règles :

**Expreb 🡪 Expreb ou Termeb OU Expreb 🡪 Termeb**

Deuxième règle

Première règle

**Exemple concret**

Représentation de la grammaire suivante :

**Représentation théorique**

S 🡪 (L) | a

L 🡪 L,S | S

**Représentation dans notre format texte**

S = ( L ) | a

L = L , S | S

***Représentation en tableaux***

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **S** |  | **=** |  | **(** |  | **L** |  | **)** |  | **|** |  | **a** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **L** |  | **=** |  | **L** |  | **,** |  | **S** |  | **|** |  | **S** |

# **Structures de données**

Nous avons utilisé deux classes afin d’encapsuler les données fournies par le fichier d’entrée. Ces classes sont : *Analyzer* et *Symbol.*

La première, ***Analyzer***, nous permet de stocker par exemple, le nom de l’axiome et la grammaire donnée par le fichier d’entrée. Elle nous permet également de regrouper de nombreuses fonctions qui permettront de trouver les premiers et les suivants d’un symbole non-terminal, ainsi que de créer la table d’analyse.

La grammaire est représentée grâce à une map. Les indexes utilisés sont de type String. Ils représentent le nom d’une variable (par exemple A, ou bien même Expression). Le contenu indexé quant à lui est un pointeur vers une instance de la classe Symbol que nous verrons juste après. C’est à partir de cette map que l’on va pouvoir accéder aux symboles non-terminaux et à leurs règles.

**Analyzer**

String \_startSymbol

Map<String, Symbol\*> \_grammar

Vector<String> \_orderedSymbols

readGrammar(..)/displayGrammar()setAllFirst()/setAllFollow()  
createTable()  
*Fonctions utiles*

**Données :**

**Fonctions :**

Le Vector de String nommé *\_orderedSymbols* stocke uniquement les noms des symboles non-terminaux dans l’ordre que de celui du fichier.

Parlons maintenant des fonctions. *readGrammar(..)* est la fonction qui va lire le fichier et stocker les données lues dans nos structures de données, tandis que *displayGrammar()* va quant à elle afficher le contenu de la map \_*grammar* tout juste remplie par *readGrammar().* setAllFirst() et setAllFollow() vont, comme leur nom l’indique, trouver et enregistrer les premiers et les suivants de tous les symboles non-terminaux stockés dans la map \_*grammar*.

La fonction *createTable()* va se servir de la map \_*grammar* pour créer la table d’analyse à partir des premiers et des suivants de chacun des symboles non-terminaux. Elle va en fait, pour chaque symbole NT, définir leur table. L’ensemble des tables des symboles NT forment la table d’analyse.

A

B

a

b

$

…

…

…

…

Table de A

Table de B

Table d’analyse

Pour ce qui est des « fonctions utiles », nous avons implémentées un certain nombre de fonction qui nous sont essentielles dans la création de la table d’analyse (donc dans la recherche des premiers et des suivants). Pour en citer quelques-unes, nous avons implémenté une fonction détectant la présence ou non du mot vide epsilon (# dans le programme). Cette fonction est très utile à de nombreuses reprises notamment lorsque l’on cherche les premiers d’un symbole non-terminal. Nous avons également une fonction nommée *findFirstIfEpsilon* qui permet d’aller chercher les autres premiers d’un symbole non-terminal si ce dernier contient un epsilon dans ses premiers. Exemple : A 🡺 BC avec B 🡺 # | a alors Premier(A) = { a ; # } mais il faut dans ce cas aller chercher les premiers de C comme B peut donner le mot vide. C’est ce que cette fonction fait.

Voyons maintenant la classe ***Symbol***.

**Symbol**

String \_name

Map<String, Vector<String>> \_table

Vector<Vector<String>> \_rules

addRule(..) / addIntoTable()  
getter/setter  
*Fonctions utiles*

**Données :**

**Fonctions :**

Vector <String> \_first

Vector <String> \_follow

Un symbole non-terminal a son nom stocké dans la variable \_*name*. La Map \_*table* est indexée par des Strings, qui représentent les symboles terminaux, tandis que le Vector de String représente quant à lui la règle de remplacement correspondante. Exemple : extrait d’une table d’analyse :

|  |  |  |
| --- | --- | --- |
|  | a | $ |
| A | A 🡺 BC | A 🡺 D |

La map \_*table* de A sera alors de la forme :

|  |  |  |  |
| --- | --- | --- | --- |
| a | |  |  | | --- | --- | | B | C | |
| $ | |  | | --- | | D | |

Le Vector de Vector de String \_*rules* représente quant à lui les règles du symbole NT. Prenons   
A 🡺 Ba | bCdD | #. On aura alors \_*rules* de la forme :

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | B | a | |
| |  |  |  |  | | --- | --- | --- | --- | | B | C | d | D | |
| |  | | --- | | # | |

Le Vector de String \_*first* et \_*follow* vont stocker respectivement les premiers et les suivants du symbole NT.

Comme bon nombre de classes, Symbol a un certain nombre de getter/setter. Elle possède notamment les fonctions *addRule*(), qui sera utilisée lorsque le programme lira le fichier d’entrée afin de définir les règles du symbole NT, ainsi que *addIntoTable*(), qui sera utilisée lorsque le programme créera la table d’analyse.

Comme la classe Analyzer, la classe Symbol possède des fonctions très pratiques, comme par exemple *isIntoTable*(…) qui va vérifier s’il existe une règle de remplacement pour un symbole NT lorsqu’il « rencontre » un symbole terminal donné (ex : on a $A (pile) a (mot à reconnaître), alors on va tester s’il existe une règle dans la table de A avec le symbole « a » ).

# **Algorithmes**

## **Lecture du fichier**

La lecture et l’interprétation du fichier texte repose sur un algorithme simple. Il suffit de lire ligne par ligne le fichier et d’analyser chaque ligne grâce à des méthodes simples.

L’analyse repose sur le principe de « split », c’est-à-dire transformer une chaine de caractères en un tableau de sous chaines de caractères en utilisant un symbole pour effectuer le découpage. Par exemple, si on split la chaine « A-B-C-D » par le symbole ‘ **-** ‘, on obtient un tableau de taille 4.



|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |

Split par ‘ – ‘

Grâce à ce procédé, on peut dégager des étapes pour effectuer l’analyse de la ligne.

Prenons comme exemple pour la suite de l’explication la ligne suivante

S = ( L ) | a

|  |  |
| --- | --- |
| **S** | **( L ) | a** |

**1. Split par ‘ = ‘**

*On a dans un tableau de taille 2 d’un côté le non terminal qui est en train d’être défini, et de l’autre ses règles qui attendent d’être analysées*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **(** | **L** | **)** | **|** | **a** |

**2. Split par ‘ ‘**

*On a split la partie de droite du tableau obtenu à l’étape1. On obtient un nouveau tableau contenant toutes les sous chaines de la ligne. Il faut maintenant traiter l’opérateur* ***|***

|  |
| --- |
| a |

|  |  |  |
| --- | --- | --- |
| ( | L | ) |

**3. Création des règles**

*Ici, on a parcouru le vector obtenu à l’étape d’avant à la recherche du symbole* ***|*** *et nous avons créé un nouveau vector de vector de strings, qui représente donc l’ensemble des règles de la grammaire*

|  |
| --- |
| a |

|  |  |  |
| --- | --- | --- |
| ( | L | ) |

**4. Vérification et traitement de la recursivité à gauche**

S

*Pour vérifier s’il y a récursivité à gauche, on parcourt le vector contenant l’ensemble des règles de la grammaire et on regarde s’il existe un premier symbole de ces règles qui est égal au symbole non terminal qui est en train d’être défini. Si c’est le cas, il y a récursivité à gauche et on lance l’algorithme d’élimination de la récursivité*

**5. Création du Symbole et ajout dans la SDD**

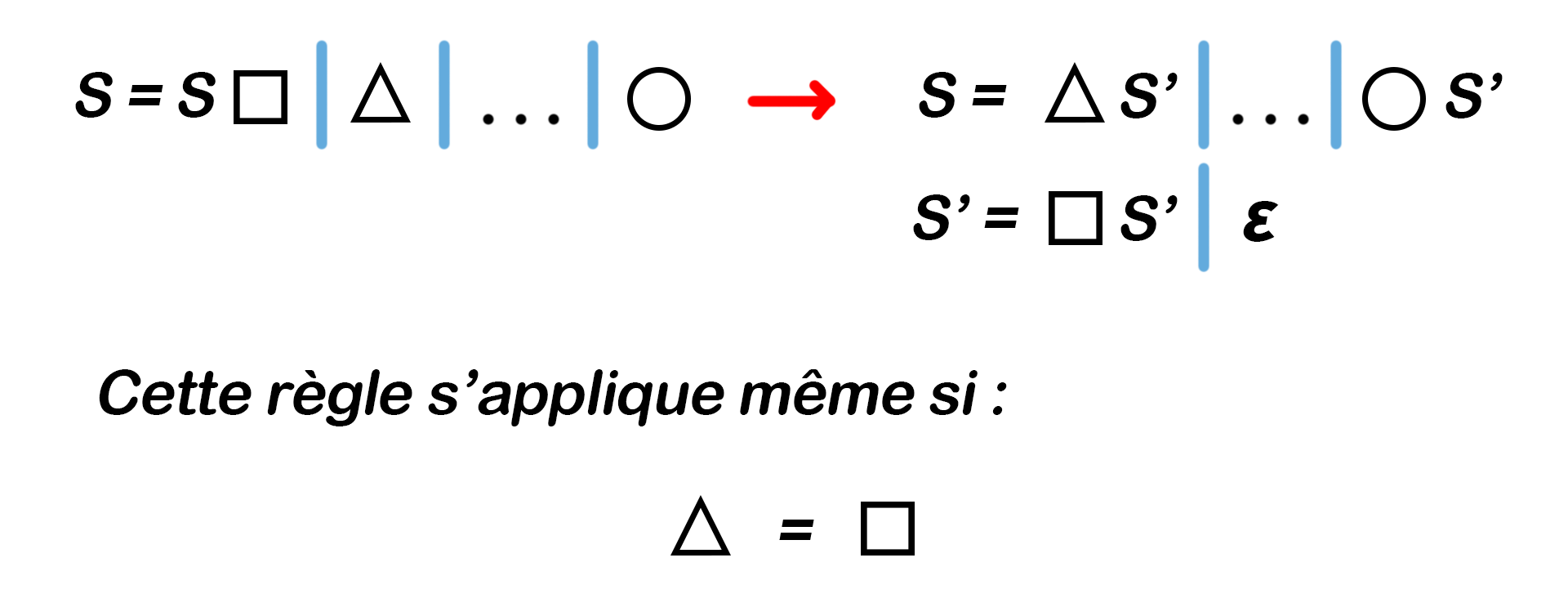
## **Elimination de la récursivité à gauche**

Lorsque l’algorithme de lecture détecte de la récursivité à gauche sur une ou plusieurs règles, il faut appliquer l’algorithme d’élimination de la récursivité.

Prenons comme symbole non terminal traité le symbole S.

On commence par créer un nouveau symbole S’, qui sera créé parallèlement à S.

On applique ensuite la règle suivante :



Il suffit juste de parcourir l’ensemble des règles du symbole S.

Pour chaque règle :

* **Récursive à gauche** 🡪 on l’ajoute dans les règles de S’, en la faisant suivre par le symbole S’
* **Pas de récursivité à gauche** 🡪 on l’ajoute dans les nouvelles règles de S, en la faisant suivre par le symbole S’

Ensuite on ajoute le mot vide dans les règles de S’, et notre symbole S ne comporte plus de récursivité à gauche.

On répète l’opération pour tous les symboles comportements de la récursivité à gauche, et nous pouvons alors passer à l’étape suivante, ajouter les symboles à la SDD.

## **Trouver les Premiers des symboles NT**

\*Cette fonction récursive findAndSetFirst(..) est très pratique puisqu’en plus de trouver et de définir les premiers d’un symbole, elle définit également les premiers de tous les symboles NT qu’elle analyse. Ex : A 🡺 C | Da | E ici en cherchant Premier(A), on aura défini : Premier(A), Premier(C), Premier(D) et Premier(E). On n’aura donc pas besoin de lancer findAndSetFirst(..) pour C, D et E.

Vérification de la grammaire

Trouver les premiers du symbole si cela n’a pas déjà été fait\*

Pour tous les symboles NT

On vérifie qu’il n’y a pas de « boucle infinie » dans la grammaire et que chaque symbole a des règles.

Exemple :

A 🡺 C | aB | bD  
B 🡺 # | a  
C 🡺 D | c  
D 🡺 A

Ici, en partant du symbole NT A, on va être amené à calculer les premiers de C puis de D pour trouver les premiers de A. Or, Premier(D) = Premier(A). Donc impossible de trouver les premiers.

Trouver les premiers d’un symbole (fonction récursive *findAndSetFirst(..)*)

Récupération de tous les symboles NT au début de chaque règle

Récupération de tous les symboles T au début de chaque règle

Ajout dans les premiers du symbole, et **on retournera la liste de tous ces symboles**

Pour tous les symboles NT, rappel de la même fonction

Ajout dans les premiers du symbole les symboles T donnés par tous les symboles NT placés au début des règles (s’il y en a)

A ce stade, on a récupéré tous les premiers du symbole

On retourne la liste des premiers du symbole

Cette liste sera réceptionnée à ce niveau dans la « couche » supérieure

Trouver les premiers (fonction *setAllFirst()*)

## **Trouver les Suivants des symboles NT**

**Parcours de tous les symboles NT de la grammaire** et vérifie si leurs suivants ont déjà été trouvés

**Suivants non trouvés** : on appelle la fonction findAndSetFollow() pour les chercher

**On cherche les suivants du symbole passé en paramètre**

**Si le symbole est un start-symbol**On **ajoute $** dans la liste des *suivants* du symbole

**On parcourt toutes les règles de la grammaire à la recherche du symbole dans la partie droite d’une règle**Si on le trouve, alors on regarde le symbole le suivant dans la règle, et on applique la méthode correspondant à chaque cas

**Si on trouve le symbole à l’intérieur d’une règle**Alors tout dépend du symbole suivant celui que l’on analyse  
- Si le symbole suivant est un terminal, alors **on l’ajoute à la liste des suivants** de notre symbole analysé.  
- Si le symbole suivant est un non-terminal, **on l’analyse à son tour** pour trouver ses premiers et les ajouter à la liste de notre symbole ; si on trouve le mot vide # parmi ces premiers, alors on cherche également ses suivants pour les ajouter à leur tour

**Si on trouve le symbole à la fin d’une règle**On cherche les suivants du symbole auquel appartient la règle

Si on cherche les suivants de B et qu’on a une règle telle que A🡪αB, alors on cherche les suivants de A pour les ajouter à ceux de B

S’il existe une production A🡪αB ou une production A🡪αBβ telle que PREMIER(β) contient *#* (c.a.d. β🡪 #), les éléments de SUIVANT(A) sont ajoutés à SUIVANT(B)

## **Créer la table d’analyse**

**On parcourt tous les symboles NT les uns après les autres**

On récupère **les règles du symbole NT actuel**, ainsi que **les symboles terminaux** commençant ces règles

**Simplification des écritures**  
Au cas où des règles commencent par le symbole #, on le supprime (par exemple B -> #C devient B-> C)

**On analyse le premier symbole de la règle**On veut déterminer si ce symbole est terminal ou non

- Si la règle est le mot vide, alors on place cette règle dans le tableau **aux coordonnées correspondant à tous les *suivants* du symbole auquel appartient la règle**(par exemple A -> #, on placera la règle dans les cases M[A,*a*] avec *a* tous les *suivants* de A )

- Si la règle n'est pas le mot vide, alors on place la règle dans le tableau **aux coordonnées qui correspondent à ce symbole terminal et au symbole auquel appartient la règle**(Pour A -> 1, on place la règle dans la case M[A,1])

**On** **regarde la liste des *premiers* de ce symbole.**

- On ajoute la règle dans la colonne de tous les symboles appartenant à la liste des *premiers* du symbole analysé et **que l'on peut trouver grâce à la règle actuelle.**

- Si on trouve le mot vide # parmi ces *premiers*, on ajoute le mot vide **pour tous les *suivants* du symbole** auquel appartient la règle.

*Si le premier symbole est terminal*

*Si le premier symbole est non terminal*

Pour chaque règle :

## **Analyser une phrase**

Récupération de la phrase à reconnaître et concaténation d’un $

On empile $ puis l’axiome

Découpage de la phrase

Phrase non reconnue ! (Arrêt)

S’il y a $ sur le haut de la pile et qu’il y a encore quelque chose à analyser dans la phrase

S’il y a # sur le haut de la pile

On dépile #

On récupère la règle de remplacement du symbole NT pour le symbole T à analyser de la phrase (s’il n’y en a pas : Phrase non reconnue ! (Arrêt))

S’il y a un symbole NT sur le haut de la pile

On dépile le symbole NT et on empile sa règle de transformation

Sinon s’il y a un symbole T sur le haut de la pile

On teste l’égalité entre le symbole T et le symbole T à analyser de la phrase

Egalité : on dépile le symbole NT, et on progresse dans la phrase à analyser

Pas d’égalité : phrase non reconnue

Tant que la pile n’est pas vide

Exemple :

Phrase : (a,(a,a))  
Concaténation : (a,(a,a))$

Découpage :

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| ( | a | , | ( | a | , | a | ) | ) | $ |

Pile :

|  |
| --- |
| A |

Progression dans la phrase : (

\_table de A :

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| ( | |  |  |  | | --- | --- | --- | | ( | B | ) | |
| … | … |

Pile :

|  |
| --- |
| ( |
| B |
| ) |

2e tour de boucle : ( est un symbole T

( == ? Progression dans la phrase : (

* Oui donc :

Pile :

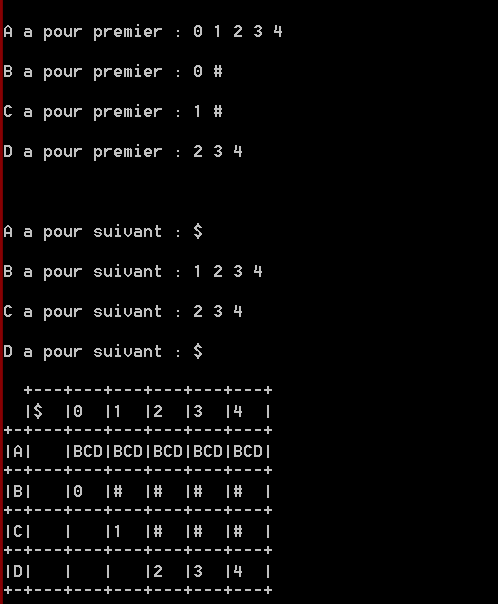
|  |
| --- |
| B |
| ) |

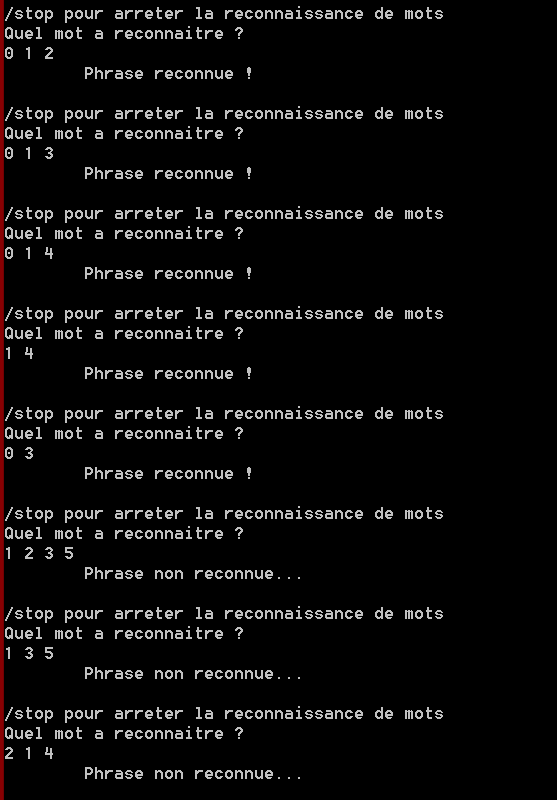
Progression dans la phrase : a  
**ETC.**

# **Traces d’exécutions**

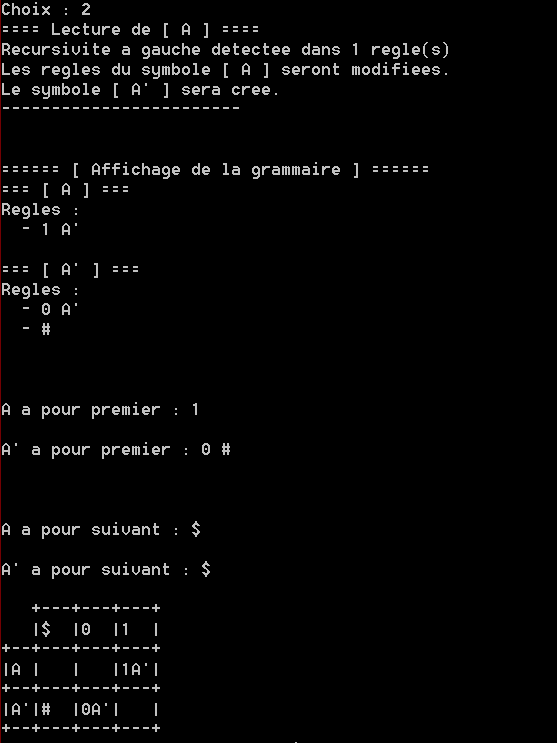
Grammaire 1 :

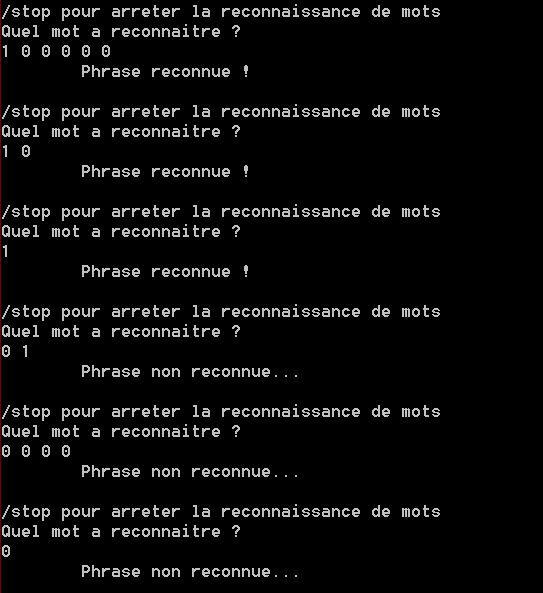






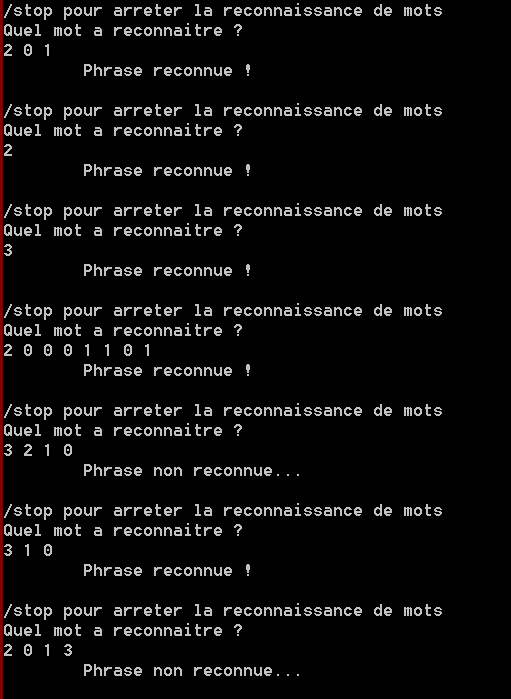
Grammaire 2:





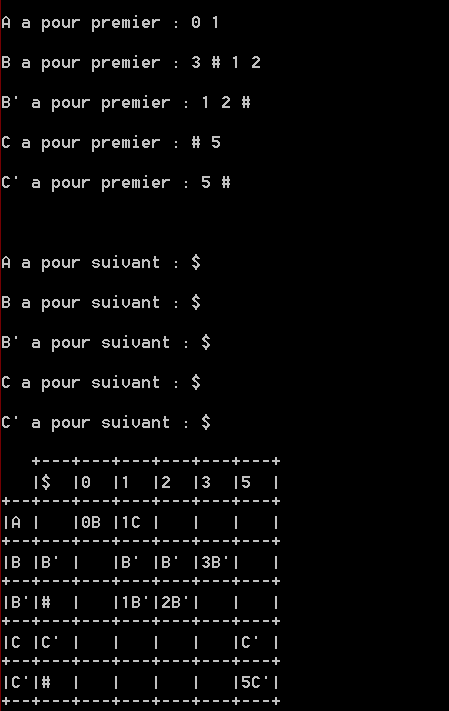
Grammaire 3:

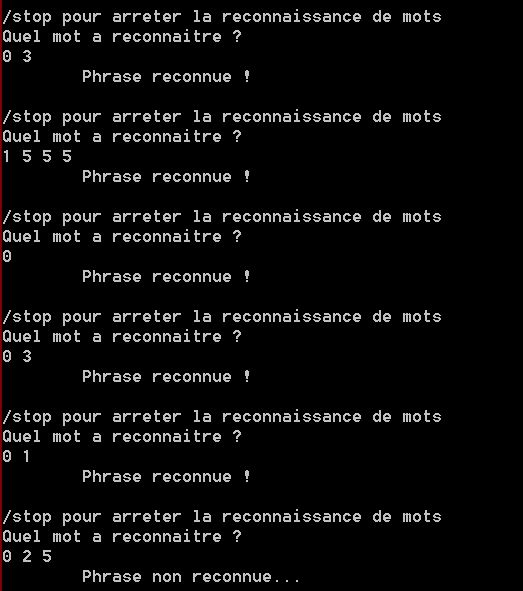




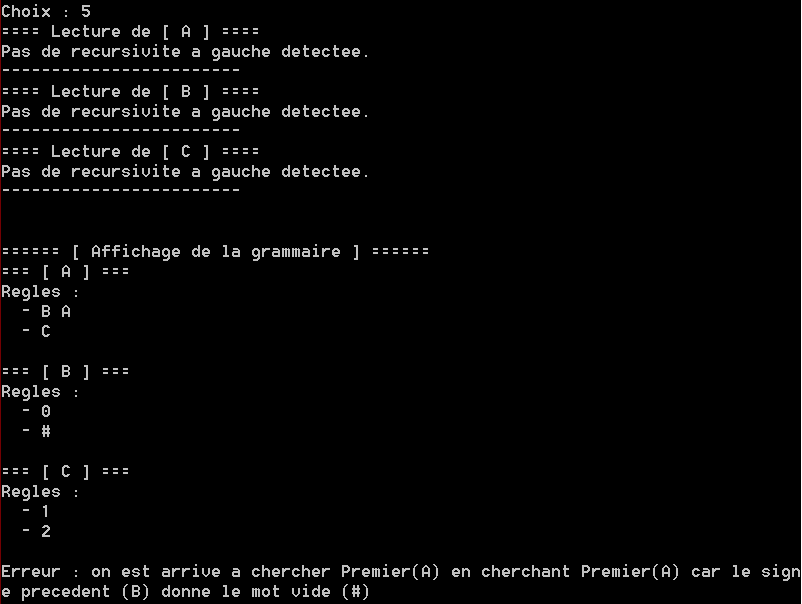
Grammaire 4:





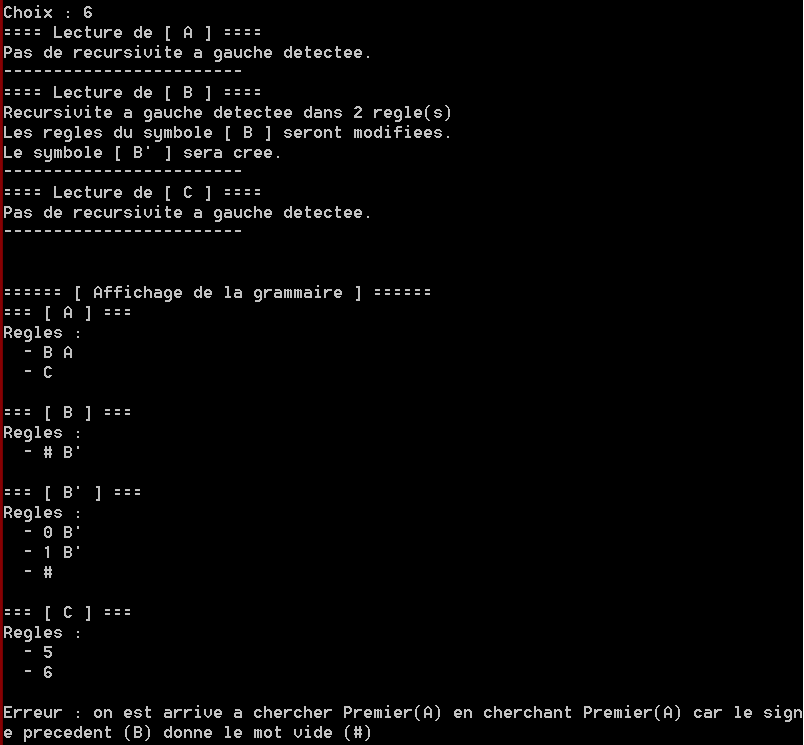


Grammaire 5:



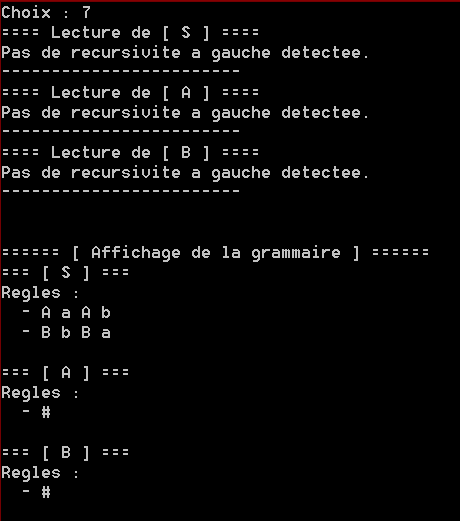
Dans ce cas, impossible de continuer.

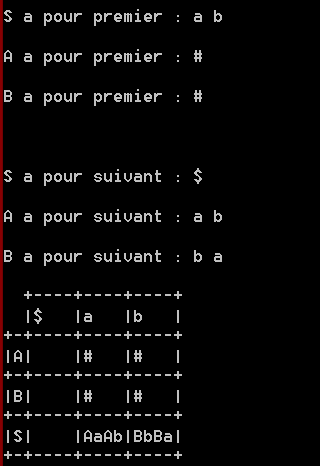
Grammaire 6:

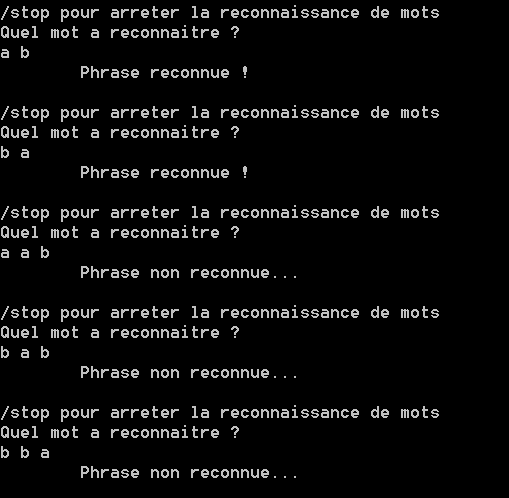


Dans ce cas, impossible de continuer.

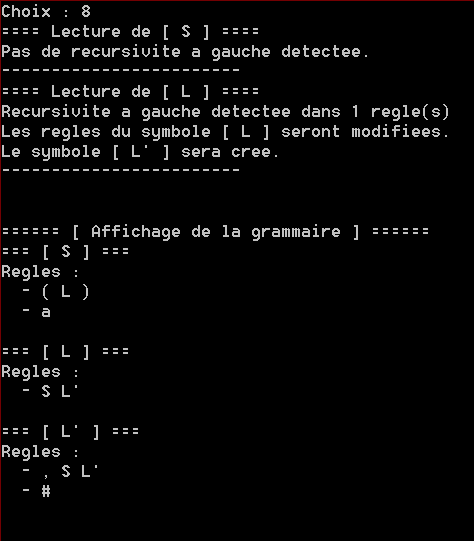
Grammaire 7:



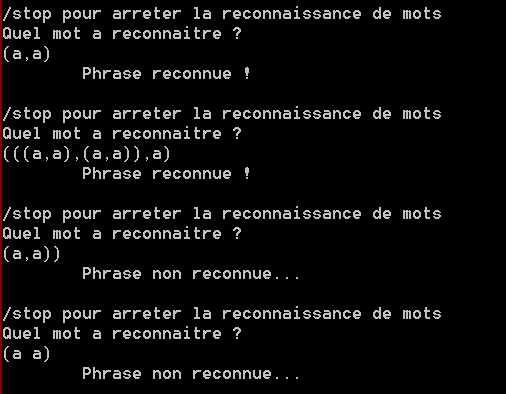




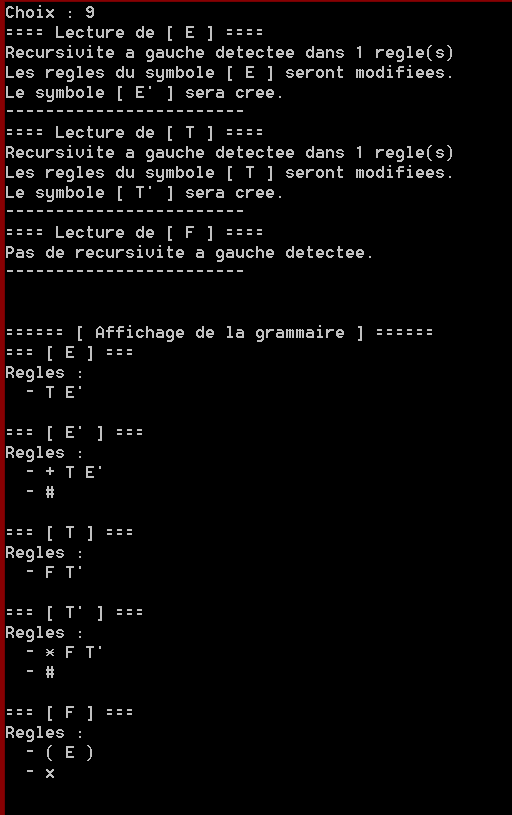
Grammaire 8:

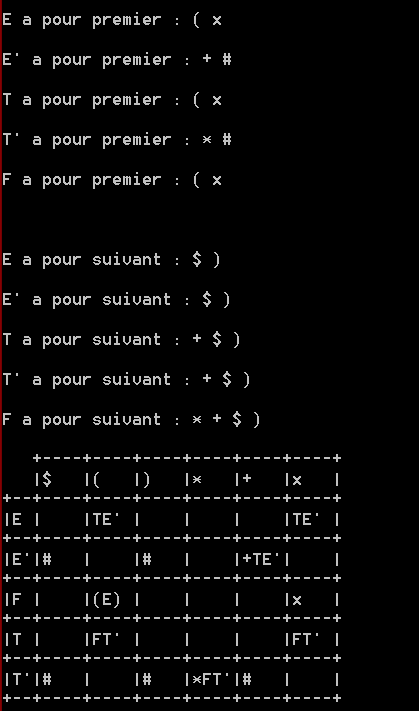


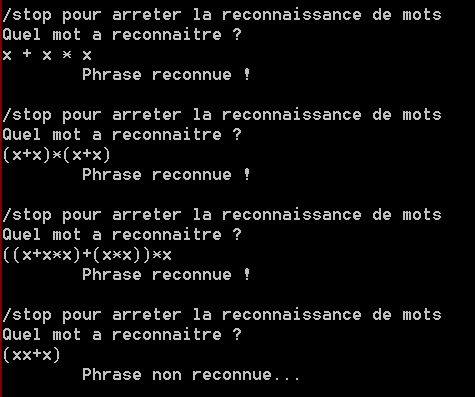




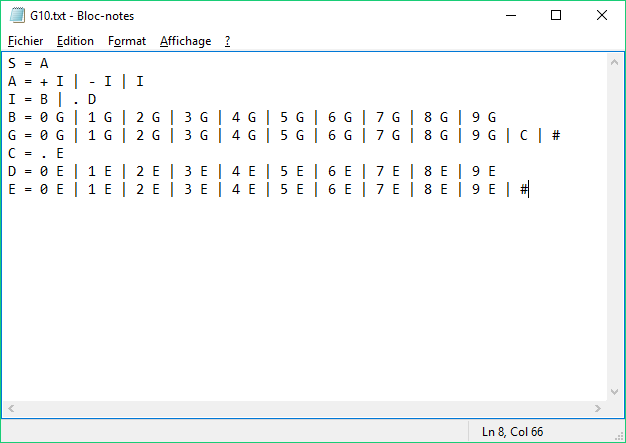
Grammaire 9:



****



Grammaire 10:



Voici ci-dessus notre grammaire pour reconnaître toutes les constantes numériques dans les langages C et C++.

Il y a beaucoup de symboles non-terminaux à cause du fait que « . » ne soit pas reconnu par le langage C/C++. Nous avions tout d’abord fait une grammaire avec 3 ou 4 symboles NT, mais avec cette grammaire, le programme était capable de reconnaître « . » comme une constante numérique. Or, le C et C++ ne le reconnaissent pas. De ce fait, il nous a fallu repenser la grammaire de façon à avoir forcément quelque chose avant et/ou après le « . » dans une constante numérique.

