Faculté des Sciences de Tunis

Rapport de TP
Techniques de Compilation
Création d'un Analyseur syntaxique et lexicale pour une partie du langage C en utilisant JAVA

Réalisé par :
Oussema HAMOUDA
Nouha ABID

IGL 3 2023/2024

Sommaire

I. Introduction

Ce rapport documente notre travail de développement d'un compilateur dédié à une portion du langage C. Notre objectif était de concevoir un outil capable d'analyser syntaxiquement et lexicalement le code source, constituant ainsi une étape fondamentale dans le processus de compilation. Ce projet a été l'occasion d'appliquer nos connaissances théoriques en théorie des langages formels et en compilation, tout en mettant en pratique nos compétences en programmation et en conception logicielle. Dans les sections suivantes, nous détaillerons les différentes phases de développement du compilateur.

II. Grammaire et notation

Dans ce qui suit on prendra en compte ces notations :

- S : axiome
- P : programme
- D : déclaration
- T : type
- I : instruction
- Ea : expression arithmétique
- V : valeur
- Ec : expression de comparaison
- Ef : expression de boucle for
- F : expression de scanf
- F1 : type d'affichage

On a choisi comme une grammaire celle présenter ci-dessous

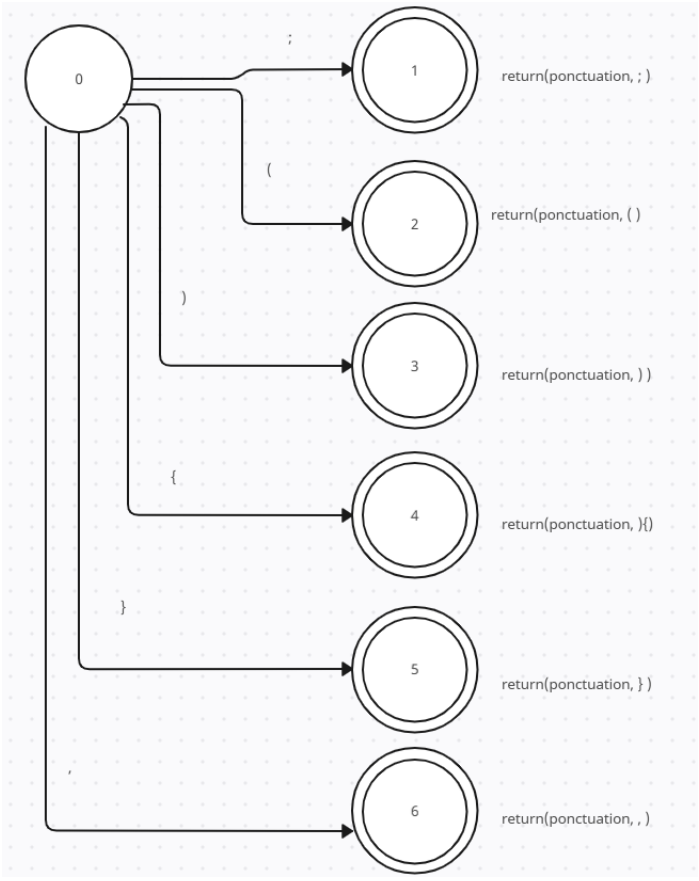
```
S -> void main ( ) { P }
P -> D ; I ;
D -> T id
D -> D ; D ;
T -> int
T -> float
T -> bool
T -> string
T -> int [ nb ]
I -> id := Ea
I -> if ( Ec ) { I }
I -> if ( Ec ) { I } else { I }
I -> for ( Ef ) { I }
I -> scanf ( F )
I -> puts ( ch )
I -> I ; I ;
Ea -> V
Ea -> ch
Ea -> V + V
```

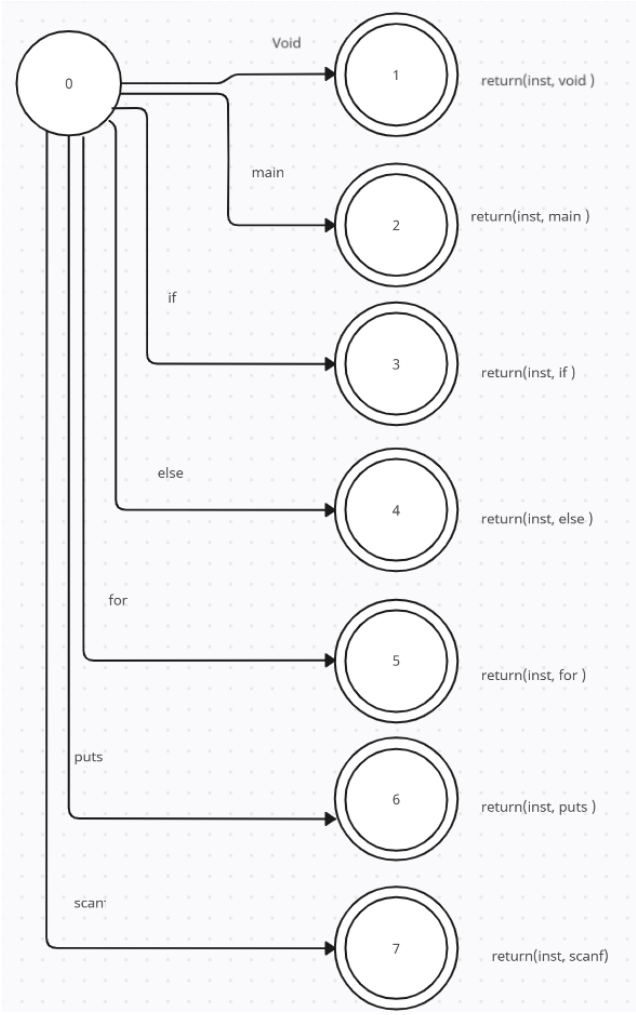
```
Ea -> V - V
Ea -> V / V
Ea -> V * V
V -> id
V -> nb
Ec -> V > V
Ec -> V = V
Ec -> V >= V
Ef -> int id := nb ; id <= nb ; id ++
F -> '% F1' , & id
F1 -> d
F1 -> f
F1 -> s
```

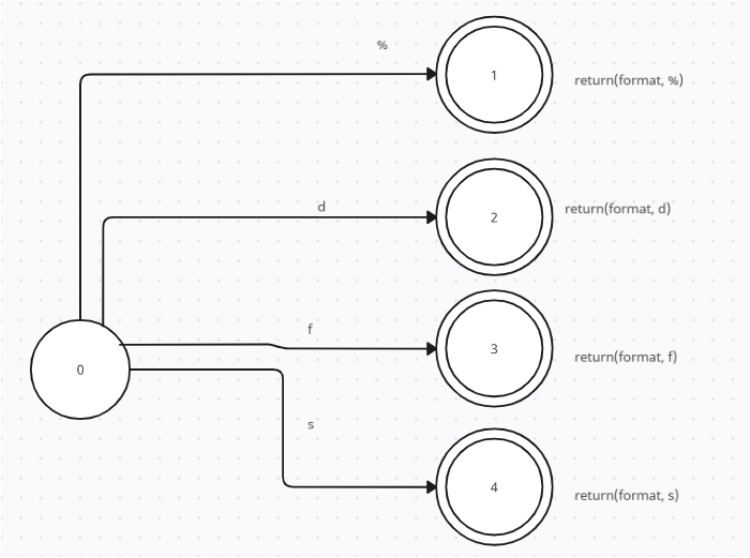
III. Analyse lexicale

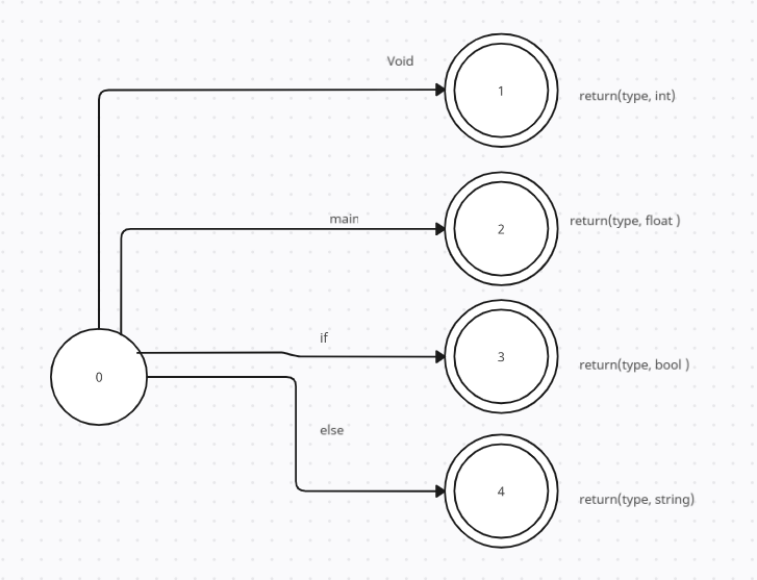
L'analyseur lexical, également appelé scanner, joue un rôle crucial dans le processus de compilation. Son travail consiste à parcourir le flux de caractères d'entrée du programme source et à le découper en unités lexicales ou tokens significatifs. Chaque token représente une unité atomique du langage de programmation, telle qu'un mot-clé, un identificateur, un opérateur, ou une constante.

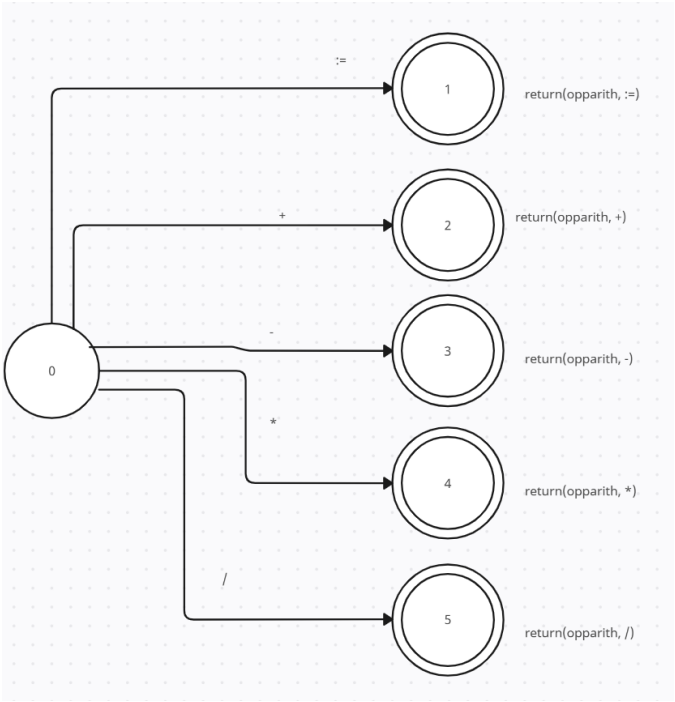
L'automate détaillé qui suit est une représentation visuelle de l'ensemble des états et des transitions que l'analyseur lexical utilise pour identifier les tokens dans le langage source.

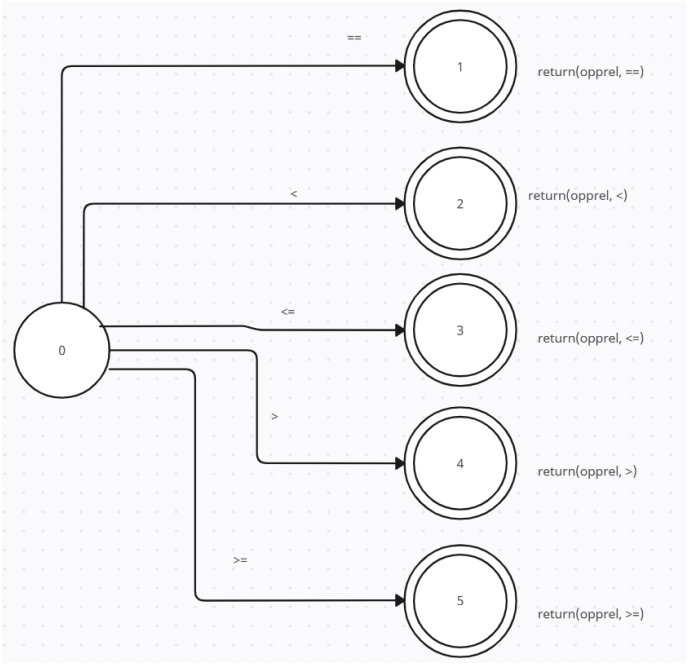


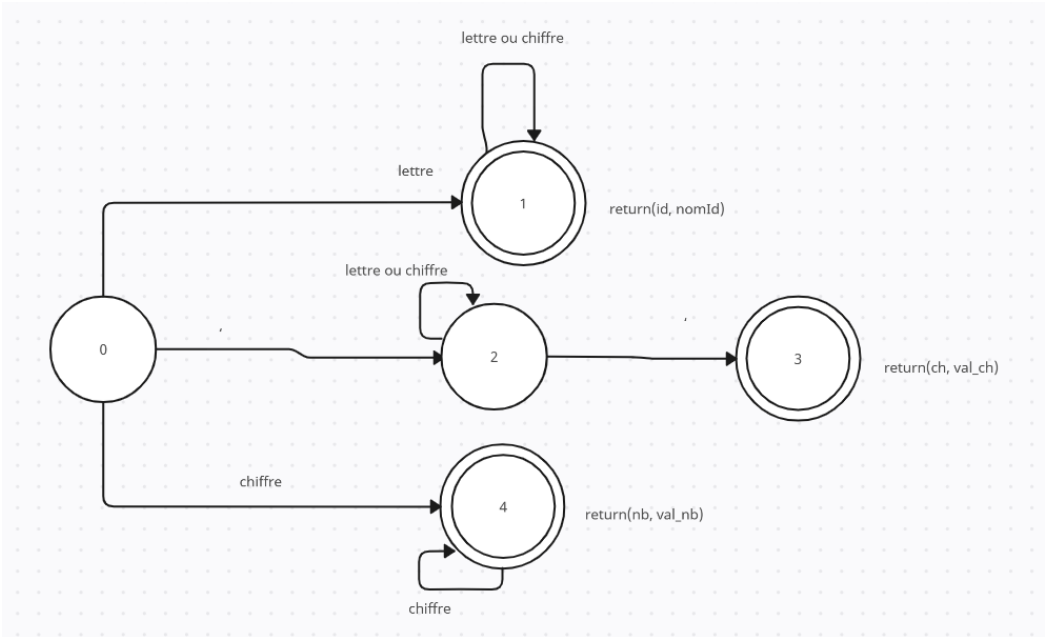



%
1
return(format, %)
d
2
return(format, d)
f
3
return(format, f)
0
s
4
return(format, s)


Void
1
return(type, int)
main
2
return(type, float)
if
3
return(type, bool)
0
else
4
return(type, string)


:=
1
return(opparith, :=)
+
2
return(opparith, +)
-
3
return(opparith, -)
0
*
4
return(opparith, *)
/
5
return(opparith, /)


==
1
return(opprel, ==)
<
2
return(opprel, <)
<=
3
return(opprel, <=)
0
>
4
return(opprel, >)
>=
5
return(opprel, >=)


lettre ou chiffre
lettre
1
return(id, nomId)
lettre ou chiffre
0
'
2
'
3
return(ch, val_ch)
chiffre
4
return(nb, val_nb)
chiffre

IV. Analyse syntaxique

Pour l'analyseur syntaxique, un terme pertinent serait "validation". L'analyse syntaxique vérifie la conformité du code source à la grammaire du langage de programmation spécifié. Ce processus de validation garantit que la structure syntaxique du programme respecte les règles définies par la grammaire, assurant ainsi sa cohérence et sa légitimité.

a.) La grammaire est-elle LL1 ?

Elimination de la récursivité à gauche

S -> void main ( ) { P }
P -> D ; I ;
D -> T id D'
D' -> ; D ; D'
D' -> ''
T -> int
T -> float
T -> bool
T -> string
T -> int [nb]
I -> id := Ea I'
I -> if (Ec) { I } I'
I -> if (Ec) { I } else { I } I'
I -> for (Ef) { I } I'
I -> scanf (F) I'
I -> puts (ch) I'
I' -> ; I ; I'
I' -> ''
Ea -> V
Ea -> ch
Ea -> V + V
Ea -> V - V
Ea -> V / V
Ea -> V * V
V -> id
V -> nb
Ec -> V > V
Ec -> V = V
Ec -> V >= V
Ef -> int id := nb ; id <= nb ; id ++
F -> '% F1' , & id
F1 -> d
F1 -> f
F1 -> s

Elimination de l'ambiguïté

S -> void main ( ) { P }
P -> D ; I ;
D -> T id D'
D' -> ; D ; D'
D' -> ''
T -> float
T -> bool
T -> string
T -> int T'
T' -> [nb]
T' -> ''
I -> id := Ea I'
I -> for (Ef) { I } I'
I -> scanf (F) I'
I -> puts (ch) I'
I -> if (Ec) { I } I''
I'' -> else { I } I'
I'' -> I'
I' -> ; I ; I'
I' -> ''
Ea -> ch
Ea -> V Ea'
Ea' -> ''
Ea' -> + V
Ea' -> - V
Ea' -> / V
Ea' -> * V
V -> id
V -> nb
Ec -> V Ec'
Ec' -> > V
Ec' -> = V
Ec' -> >= V
Ef -> int id := nb ; id <= nb ; id ++
F -> '% F1' , & id
F1 -> d
F1 -> f
F1 -> s

Les premiers et les suivants :

Nonterminal	Premier	Suivant
S	{void}	{$}
P	{int,float,bool,string}	{}}
D	{int,float,bool,string}	{;}
D'	{;,''}	{;}
T	{int,float,bool,string}	{id}
T'	{[,''}	{id}
I	{id,if,for,scanf,puts}	{;,}}
I'	{;,''}	{;,}}
I''	{;,'',else}	{;,}}
Ea	{ch,id,nb}	{;,}}
Ea'	{+,-,/,*,''}	{;,}}
V	{id,nb}	{+,-,/,*,;,>,=,>=,),}}
Ec	{id,nb}	{)}
Ec'	{>,=,>=}	{)}
Ef	{int}	{)}
F	{'}	{)}
F1	{d,f,s}	{'}

Table d'analyse LL1 :

(la totalité de la table est disponible dans le fichier Excel compilation TP, inclus dans le dossier Drive)

		}	;	id	int	float	bool	string
S								
P					P -> D ; I ;	P -> D ; I ;	P -> D ; I ;	P -> D
D					D -> T id D'	D -> T id D'	D -> T id D'	D -> T
D'			D' -> ; D ; D' D' -> ε					
T					T -> int T'	T -> float	T -> bool	T -> st
T'				T' -> ε				
I				I -> id = Ea I'				
I'		I' -> ε	I' -> ; I ; I' I' -> ε					
I''		I'' -> I'	I'' -> I'					
Ea				Ea -> V Ea'				
Ea'		Ea' -> ε	Ea' -> ε					
V				V -> id				
Ec				Ec -> V Ec'				
Ec'								
Ef					Ef -> int id = nb ; id <= nb ; id ++			
F								
F1								

On remarque qu'il y a des doublants c-à-d des cases contenant 2 règles de production
=> Donc la grammaire n'est pas de type LL1

b.) Analyse SLR

Augmentation

S' -> S
S -> void main ( ) { P }

Les transitions

(la totalité du tableau est disponible dans le fichier Excel compilation TP, inclus dans le dossier Drive)

Transiter		état	Fermeture
	{S' -> .S}	0	{S' -> .S; S -> .void main ( ) { P }}
Transiter(0, S)	{S' -> S.}	1	{S' -> S.}
Transiter(0, void)	{S -> void.main ( ) { P }}	2	{S -> void.main ( ) { P }}
Transiter(2, main)	{S -> void main.( ) { P }}	3	{S -> void main.( ) { P }}
Transiter(3, ()	{S -> void main (.) { P }}	4	{S -> void main (.) { P }}
Transiter(4,))	{S -> void main ( ).{ P }}	5	{S -> void main ( ).{ P }}
Transiter(5, {)	{S -> void main ( ) {.P }}	6	{S -> void main ( ) {.P }; P -> .D ; I ;; D -> .T id; D -> .D ; D ;; T -> .int; T -> .fl
Transiter(6, P)	{S -> void main ( ) { P.}}	7	{S -> void main ( ) { P.}}
Transiter(6, D)	{P -> D.; I ;; D -> D.; D ;}	8	{P -> D.; I ;; D -> D.; D ;}
Transiter(6, T)	{D -> T.id}	9	{D -> T.id}
Transiter(6, int)	{T -> int.; T -> int.[nb]}	10	{T -> int.; T -> int.[nb]}
Transiter(6, float)	{T -> float.}	11	{T -> float.}
Transiter(6, bool)	{T -> bool.}	12	{T -> bool.}
Transiter(6, string)	{T -> string.}	13	{T -> string.}
Transiter(7, })	{S -> void main ( ) { P }.}	14	{S -> void main ( ) { P }.}
Transiter(8, ;)	{P -> D ;.I ;; D -> D ;.D ;}	15	{P -> D ;.I ;; D -> D ;.D ;; I -> .id := Ea; I -> .if (Ec) { I }; I -> .if (Ec) { I } else
Transiter(9, id)	{D -> T id.}	16	{D -> T id.}
Transiter(10, [)	{T -> int [.nb]}	17	{T -> int [.nb]}
Transiter(15, I)	{P -> D ; I.;; I -> I.; I ;}	18	{P -> D ; I.;; I -> I.; I ;}
Transiter(15, D)	{D -> D ; D.;; D -> D.; D ;}	19	{D -> D ; D.;; D -> D.; D ;}
Transiter(15, id)	{I -> id.:= Ea}	20	{I -> id.:= Ea}
Transiter(15, if)	{I -> if.(Ec) { I }; I -> if.(Ec) { I } else { I }}	21	{I -> if.(Ec) { I }; I -> if.(Ec) { I } else { I }}
Transiter(15, for)	{I -> for.(Ef) { I }}	22	{I -> for.(Ef) { I }}

Table d'analyse SLR

(la totalité de la table est disponible dans le fichier Excel compilation TP, inclus dans le dossier Drive)

	void	main	(	)	{	}	;	id	int	float	bool
0	s2	err	err	err	err	err	err	err	err	err	err
1	err	err	err	err	err	err	err	err	err	err	err
2	err	s3	err	err	err	err	err	err	err	err	err
3	err	err	s4	err	err	err	err	err	err	err	err
4	err	err	err	s5	err	err	err	err	err	err	err
5	err	err	err	err	s6	err	err	err	err	err	err
6	err	err	err	err	err	err	err	err	s10	s11	s12
7	err	err	err	err	err	s14	err	err	err	err	err
8	err	err	err	err	err	err	s15	err	err	err	err
9	err	err	err	err	err	err	err	s16	err	err	err
10	err	err	err	err	err	err	err	r5	err	err	err
11	err	err	err	err	err	err	err	r6	err	err	err
12	err	err	err	err	err	err	err	r7	err	err	err
13	err	err	err	err	err	err	err	r8	err	err	err
14	err	err	err	err	err	err	err	err	err	err	err
15	err	err	err	err	err	err	err	s20	s10	s11	s12
16	err	err	err	err	err	err	r3	err	err	err	err
17	err	err	err	err	err	err	err	err	err	err	err
18	err	err	err	err	err	err	s26	err	err	err	err
19	err	err	err	err	err	err	s27	err	err	err	err
20	err	err	err	err	err	err	err	err	err	err	err
21	err	err	s29	err	err	err	err	err	err	err	err
22	err	err	s30	err	err	err	err	err	err	err	err

V.Exemple d'exécution

Voici un exemple d'exécution valide pour la chaîne :

(Vous pouvez trouver le résultat de l'exécution dans le fichier exemple.txt inclus dans le dossier Drive)

```
void main ( ) { float x ; x := x + 10  ; puts ( 'HI' ) ; ; if ( x > 5) { puts ( 'sup' ) } else { puts ( 'inf' ) }  ; ; }
```

```
taper votre texte ci-dessous (ctrl+z pour finir)
void main ( ) { float x ; x := x + 10  ; puts ( 'HI' ) ; ; if ( x > 5) { puts ( 'sup' ) } else { puts ( 'inf' ) }  ; ; }
^Z
**********Analyse lexical**********
inst    void
inst    main
ponctuation     (
ponctuation     )
ponctuation     {
type_var        float
id      x       0#
ponctuation     ;
id      x       0#
oparith :=
id      x       0#
oparith +
nb      10
ponctuation     ;
inst    puts
ponctuation     (
ch      HI
ponctuation     )
ponctuation     ;
ponctuation     ;
inst    if
ponctuation     (
id      x       0#
oprel   >
nb      5
ponctuation     )
ponctuation     {
inst    puts
ponctuation     (
ch      sup
ponctuation     )
ponctuation     }
inst    else
ponctuation     {
inst    puts
ponctuation     (
ch      inf
ponctuation     )
ponctuation     }
ponctuation     ;
ponctuation     ;
ponctuation     }
eof     0
```

```
analyze SLR successfully
```

Voici un exemple d'exécution avec erreur pour la chaîne :

(Vous pouvez trouver le résultat de l'exécution dans le fichier exemple_avec_erreur.txt inclus dans le dossier Drive)

```
void main ( ) { int 1x ; puts ( 'hi' ) ; }
```

```
taper votre texte ci-dessous (ctrl+z pour finir)
void main ( ) { int 1x ; puts ( 'hi' ) ; }
^Z
**********Analyse lexical**********
inst    void
inst    main
ponctuation     (
ponctuation     )
ponctuation     {
type_var        int
nb      1
id      x       0#
ponctuation     ;
inst    puts
ponctuation     (
ch      hi
ponctuation     )
ponctuation     ;
ponctuation     }
eof     0
**********Analyse Syntaxique**********
********pile     Entrée   Action
[0]----------voidmain(){intnbid;puts(ch);}$------
-
[0, void, 2]----------main(){intnbid;puts(ch);}$-
-------------shift
[0, void, 2, main, 3]----------(){intnbid;puts(ch
);}$--------------------shift
[0, void, 2, main, 3, (, 4]----------){intnbid;pu
ts(ch);}$---------------------------shift
[0, void, 2, main, 3, (, 4, ), 5]----------{intnb
id;puts(ch);}$----------------------------------shift
[0, void, 2, main, 3, (, 4, ), 5, {, 6]----------
intnbid;puts(ch);}$-----------------------------------shift
[0, void, 2, main, 3, (, 4, ), 5, {, 6, int, 10]----------
---------nbid;puts(ch);}$-----------------------------------------shift
texterreurerr10nb6
analyze SLR failled
```

VI. Conclusion

En conclusion, le présent rapport témoigne du parcours entrepris dans le développement d'un compilateur pour une portion du langage C, mettant en œuvre des analyses syntaxique et lexicale. À travers ce projet, nous avons navigué avec rigueur et engagement, depuis la conception de la grammaire jusqu'à l'implémentation des tableaux d'analyses SLR, en veillant à garantir la cohérence et la précision des règles syntaxiques établies. Bien que notre travail puisse être considéré comme une étape initiale dans le développement d'un compilateur complet pour le langage C, il représente néanmoins une réalisation significative qui témoigne de notre détermination à relever des défis complexes et à pousser les limites de notre expertise.