المملكة المغربية
Royaume du Maroc
Ecole Hassania des Travaux Publics
المدرسة الحسنية للأشغال العمومية

Projet Compilation

Langage C--

2018/2019

Réalisé Par :

LIMOURI Anas

MAADA Loukmane

MATHANI Abdessamad

MOUMOU Soukaina

Encadré par :

Mr. ELGHAZI Souhail

Table des matières

Introduction

Ce livrable constitue la partie lexicale et la partie syntaxique de notre projet qui est la réalisation d'un compilateur du langage C--, dans cette étape on va préciser l'alphabet du langage, ses mots clés, sa grammaire améliorée, la table des unités lexicales ainsi que l'automate.

Partie lexicale

Alphabet et mots clés

L'alphabet du langage C-- est :

∑ = { , ,! , & , (,) , * , + , - , . , / , { , | , } [,] , _ , ; , < , = , > , 0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9 , A , B , C , D , E , F , G , H , I , J , K , L , M , N , O , P , Q , R , S , T , U , V , W , X , Y , Z , a , b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r , s , t , u , v , w , x , y , z }

Le langage C-- contient la liste des mots clé suivants :

Mots _ Clé = { extern, void, int, for, while, if, then, else, struct}

Struct : suite à l'amélioration que nous allons apporter au langage C--, nous envisageons un nouveau mot clé qui donnera la possibilité de déclarer un struct.

Exemple de programme :

```
int macro = 0 ;
int somme(int x, int y){
return x + y ;
}
Int main(){
int variableLocal = 6 ;
int variableResultat = somme(variableLocal, macro) ;
}
```

Grammaire de base améliorée

<programme> : <liste-declaration><liste-fonctions>

<liste-declaration> : <liste-declaration><declaration> | epsilon

<liste-fonctions> : <liste-fonctions> <fonction> | epsilon

<declaration> : int <liste-declarations> ;

<liste-declarateurs> : <liste-declarateurs> , <declarateur> | <declarateur>

<declarateur> : identificateur | identificateur [constante] | <struct>

<fonction> : <type> identificateur (<liste-params>) { <liste-declarations> <liste-instructions>} | extern <type> identificateur (<liste-params>) ;

<type> : void | int

<liste-params> : <liste-params> , <param> | epsilon

<param> : int identificateur

<liste-instructions> : <liste-instructions> <instruction> | epsilon

<instruction> : <iteration> | <selection> | <saut> | <affectation> ; | <bloc> | <appel>

<iteration> : for (<affectation> ; <condition> ; <affectation>) <instruction> | while (<condition>) <instruction>

<selection> : if (<condition>) <instruction> | if (<condition>) <instruction> else <instruction>

<saut> : return ; | return <expression> ;

<affectation> : <variable> = <expression>

<bloc> : { <liste-instructions>}

<appel> : identificateur (<liste-expressions>) ;

<variable> : identificateur | identificateur [<expression>] | identificateur . identificateur

<expression> : (<expression>) | <expression> <binary-op> <expression> | - <expression> | <variable> | identificateur (<liste-expressions>)

<liste-expressions> : <liste-expressions> , <expression> | epsilon

<condition> : ! (<condition>) | <condition> <binary-rel> <condition> |
(<condition>) |
<expression> <binary-comp> <expression>

<binary-op> : + | - | * | / | << | >> | & | | |

<binary-rel> : && | ||

<binary–comp> : < | > | >= | <= | == | !=

Amélioration :

Notre groupe a eu la tâche d'ajouter *la structure* à la grammaire. En orange sont les champs ajoutés pour adopter *la structure en C* à la grammaire existante.

<struct> : struct identificateur { <liste-attributs> } ; | struct identificateur identificateur <liste-declarateurs-struct>

<liste attributs> : int <declarateur> ; <liste-attributs> | int <declarateur> ;

<liste-declarateurs-struct> : , identificateur <liste-declarateurs-struct> | epsilon

Structure du lexical et choix techniques :

Notre programme se constitue de 3 fichiers :

- Lexical.h :

Ce fichier contient la définition de la classe Lexical qui contient les structures de données et les méthodes nécessaires au fonctionnement de l'analyseur lexical notamment les tables de hachage des mots clé et des identificateurs la méthode ***T_Lexeme uniteSuivante(),*** les fonctions de hachage, etc.

- Constantes.h :

Ce fichier contient la définition de la table des mots clés.

- Structures.h :

Contient les énumérations des unités lexicales et la structure des unités lexicales.

- Lexical.cpp :

Ce fichier contient l'implémentation des différentes méthodes de la classe Lexical.

- Main.cpp :

Dans ce fichier on teste le fonctionnement de l'analyseur lexical.

Choix Techniques :

- On a utilisé deux tables de hachage :

Une pour les mots clé et l'autre pour les identificateurs afin de faciliter l'accès à ces données.

Pour les fonctions de hashages, on a utilisé deux fonctions faciles.

HashMotCle : Fonction qui retourne la somme des codes ASCII de chaque caractère de la chaine multiplié par sa position dans la chaine. Elle garantit l'unicité du hashage de chaque mot clé dans notre cas.

HashIdent : Fonction qui hashe les identifiants, cependant, cette dernière n'assure pas l'unicité c'est pourquoi on a utilisé une table de hachage sous forme de tableau de vecteurs de taille dynamique.

Table des unités lexicales

Les symboles mentionnés dans le tableau des unités lexicales :

lettre = { A , B , C , D , E , F , G , H , I , J , K , L , M , N , O , P , Q , R , S , T , U , V , W , X , Y , Z , a , b , c , d , e , f , g , h , i , j , k , l , m , n , o , p , q , r , s , t , u , v , w , x , y , z}

chiffre = {0 , 1 , 2 , 3 , 4 , 5 , 6 , 7 , 8 , 9}

Lexème	***Unité Lexicale***	***Attribut***	***Modèle***
x1, var, var_2…	IDENT	Entier (Position dans la table des identificateurs)	(lettre)(lettre |_| chiffre)*
11, 1234, 1…	CONST_INT	Entier (sa valeur)	(chiffre)|(chiffre)*
'if++', 'while'….	MOT_CLE	Entier (Position dans la table des mots clé)	(lettre)+
','	VIRG	--	,
' !'	NON	--	!
'+'	PLUS	--	+
'-'	MOINS	--	-
'*'	MULT	--	*
'/'	DIV	--	/
'<<'	DEC_DROITE	--	<<
'>>'	DEC_GAUCHE	--	>>
'&'	BIN_ET	--	&
'|'	BIN_OU	--	|
'&&'	ET	--	&&
'||'	OU	--	||
'<'	INF	--	<
'>'	SUP	--	>
'<='	INFEGAL	--	<=
'>='	SUPEGAL	--	>=
'=='	EGAL	--	==
'! ='	DIFF	--	!=
'='	AFFEC	--	=
' ;'	SEP	--	;
'('	PAR_OUV	--	(
')'	PAR_FER	--	)
'*'	CRO_OUV	--	[

‘+’	CRO_FER	--	]
‘{‘	ACC_OUV	--	{
‘} ‘	ACC_FER	--	}
‘.’	POINT	--	.
‘EOF ‘	ENDOF	--	EOF

Table des mots clés

Mot Clé	*Attribut*
If	59
then	67
else	66
while	79
for	166
extern	73
int	173
void	55
struct	98
return	112

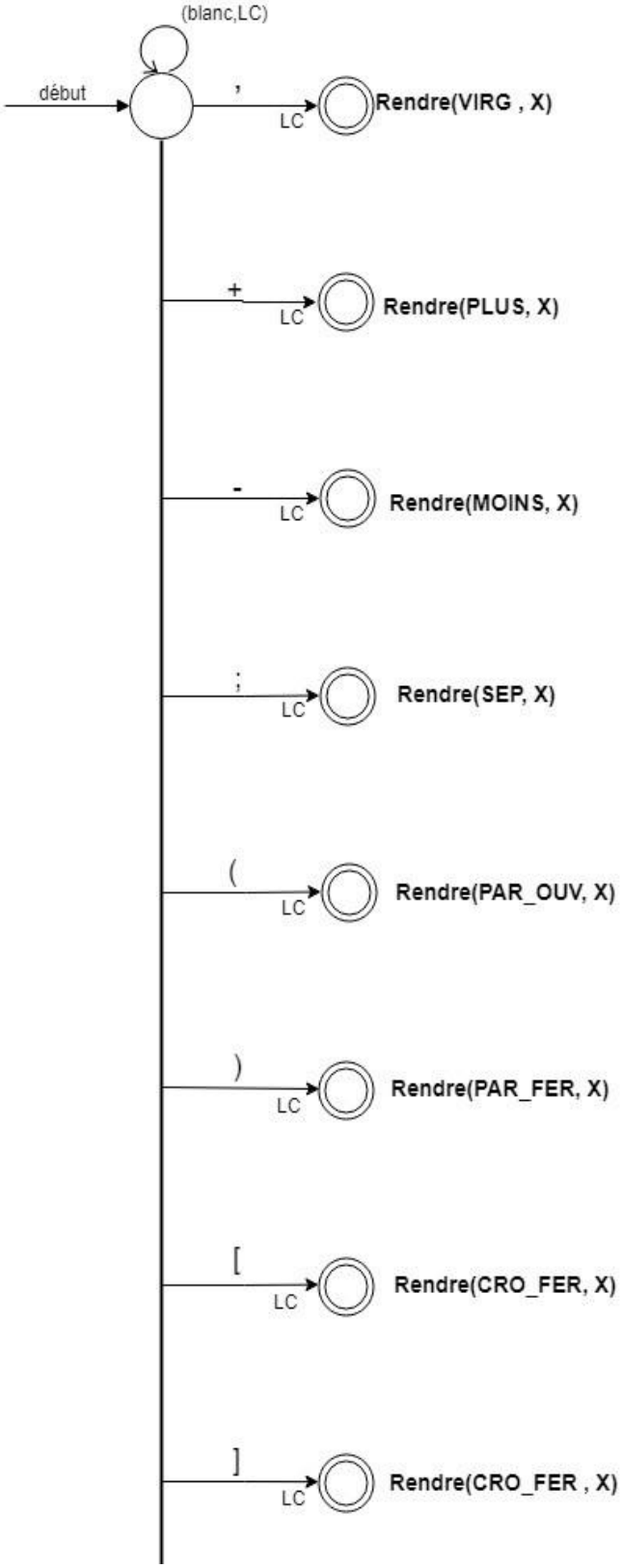
Tables des erreurs

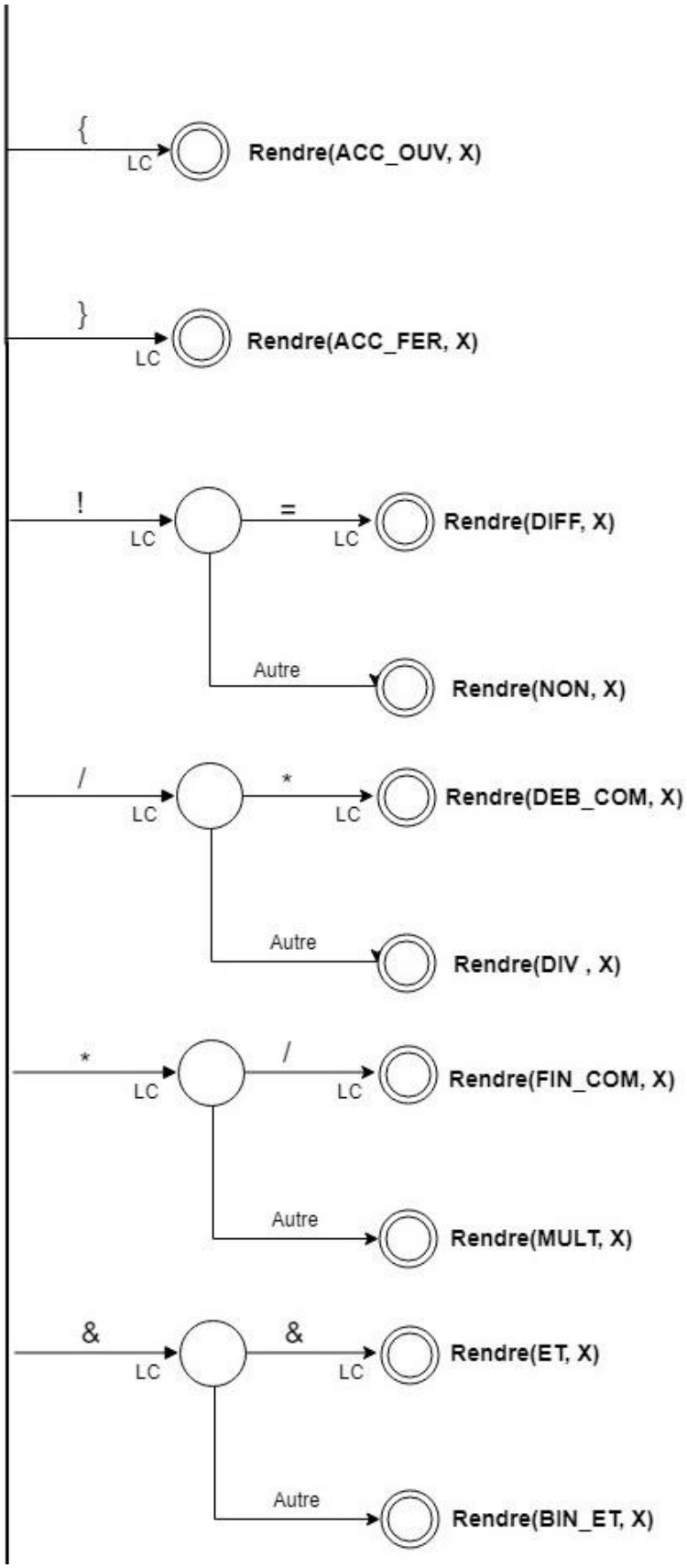
Erreur	*Son code*
Une variable ne commence pas par un « underscore »	1
Les caractères utilisés ne sont pas reconnus	2

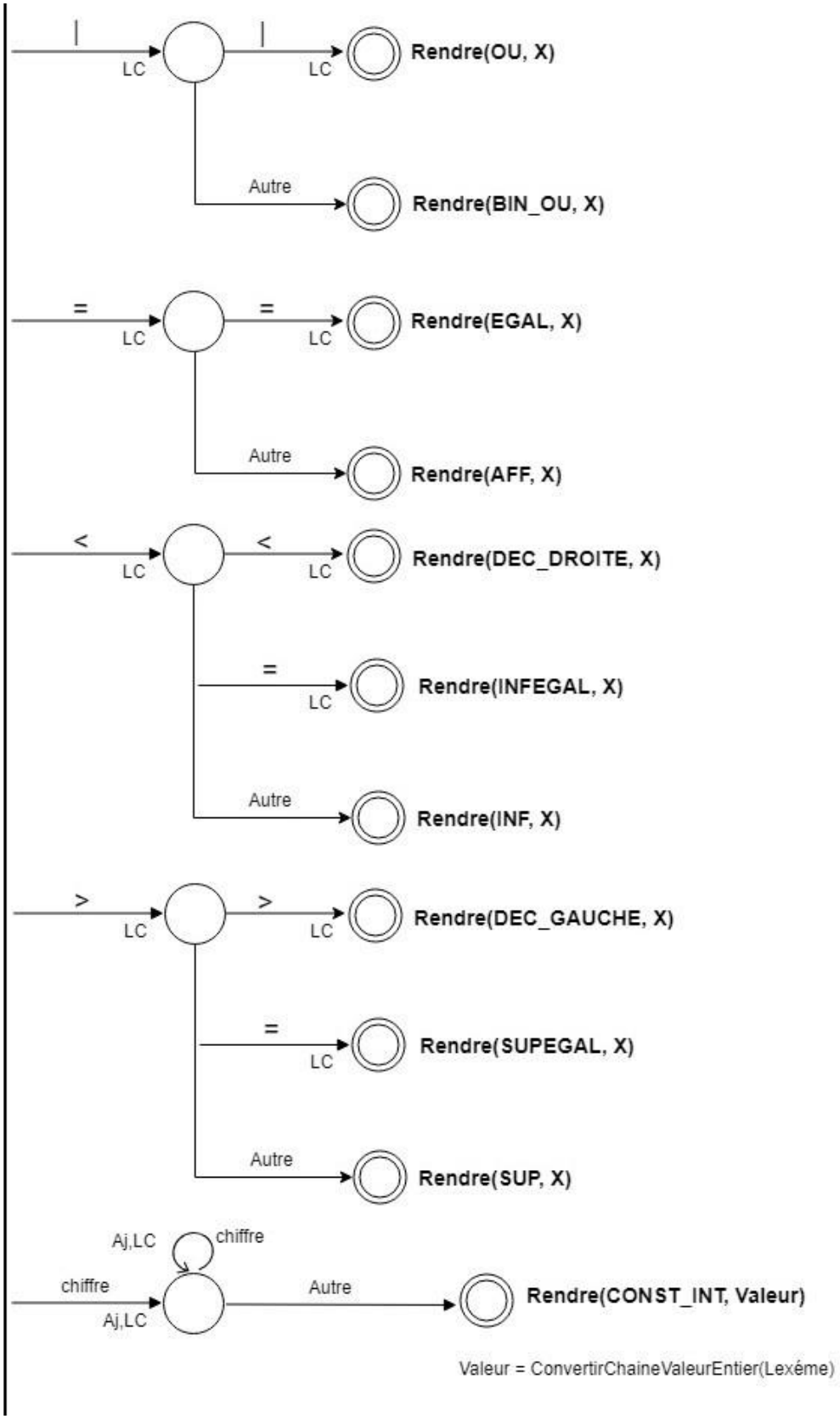
Exemple table identifiant correspondant à l’exemple du code:

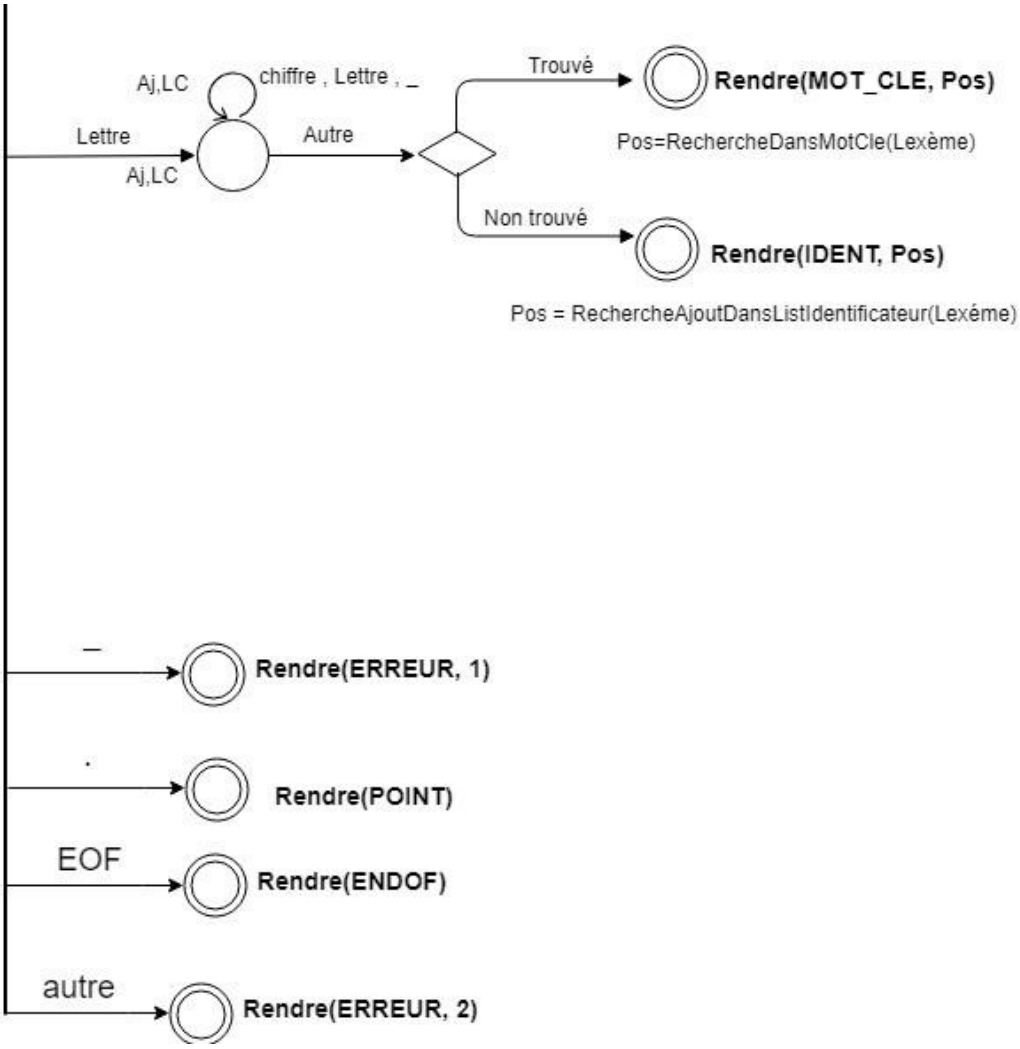
Identifiant	*Attribut*
macro	0
somme	1
variableLocale	2
VariableResultat	3

Automate à états finis




{
LC
Rendre(ACC_OUV, X)
}
LC
Rendre(ACC_FER, X)
!
LC
=
LC
Rendre(DIFF, X)
Autre
Rendre(NON, X)
/
LC
*
LC
Rendre(DEB_COM, X)
Autre
Rendre(DIV , X)
*
LC
/
LC
Rendre(FIN_COM, X)
Autre
Rendre(MULT, X)
&
LC
&
LC
Rendre(ET, X)
Autre
Rendre(BIN_ET, X)


|
LC
|
LC
Rendre(OU, X)
Autre
Rendre(BIN_OU, X)
=
LC
=
LC
Rendre(EGAL, X)
Autre
Rendre(AFF, X)
<
LC
<
LC
Rendre(DEC_DROITE, X)
=
LC
Rendre(INFEGAL, X)
Autre
Rendre(INF, X)
>
LC
>
LC
Rendre(DEC_GAUCHE, X)
=
LC
Rendre(SUPEGAL, X)
Autre
Rendre(SUP, X)
Aj,LC
chiffre
chiffre
Aj,LC
Autre
Rendre(CONST_INT, Valeur)
Valeur = ConvertirChaineValeurEntier(Lexéme)



Fonctions utilisées :

int ConvertirChaineValeurEntier(char* lexème) : convertie le chiffre d'une chaine de caractère à l'entier correspondant.

int RechercheDansMotCle(char* lexème) : cherche le lexème dans la table des mots clé et retourne son indice s'il le trouve sinon retourne -1.

int RechercheAjoutDansListIdentificateur(char* lexème) : cherche le lexème dans la table des identifiant, l'ajoute s'il n'y figure pas, puis revoie son indice.

Difficultés rencontrées (Lexical)

- Eviter le conflit causé par le Hashage dans le cas des identificateurs : Notre conception de la table des identificateurs utilise une table de vecteurs. Chaque case du tableau contient un vecteur où on stocke les identificateurs qui ont le même Hashcode, afin d'éviter les collisions. Cette structure implique que le retour de la fonction uniteSuivante() doit avoir trois attributs :
    - Unité lexicale correspondante au lexème lu
    - Indice dans le tableau des identificateurs
    - Indice au sein du vecteur dynamique

- Cette nouvelle structure imposera un changement dans la conception initiale de la structure TuniteLgoglexicale qui deviendra :

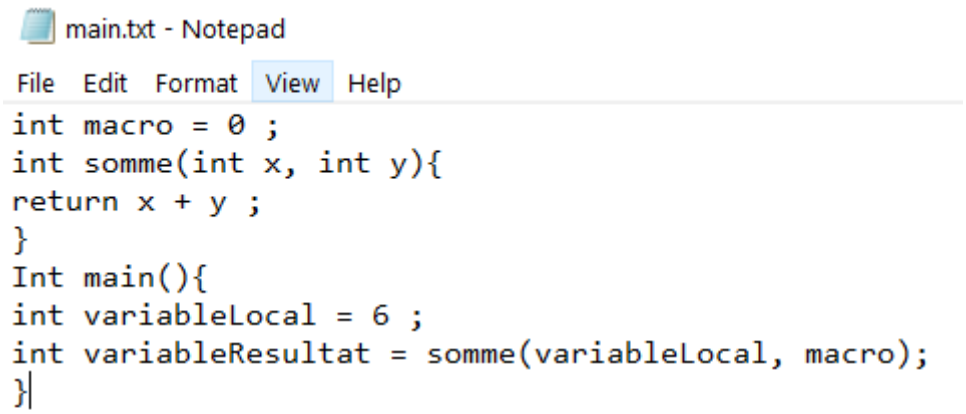
```
*Typedef struct {
        Tunite lexeme ;
        int attribut ;
        int attributConflit ; } TuniteLexicale ;
```

- Contrainte de la salle : nous avons du mal à trouver où se réunir pour travailler sur le projet, surtout les week-ends, vu que la salle 2-GI n'est disponible qu'après 19h.

Jeux de scénario

Dans cette section, nous allons exposer des exemples de programmes et l'exécution du Lexical sur ceux-ci.

Exemple de code



Exécution du programme

```
========== Output Lexical ===========
(MOT_CLE, 173)
(IDENT, 45542, 0, macro)
AFFEC
(CONST_INT, 0)
SEP
(MOT_CLE, 173)
(IDENT, 64881, 0, somme)
PAR_OUV
(MOT_CLE, 173)
(IDENT, 120, 0, x)
VIRG
(MOT_CLE, 173)
(IDENT, 121, 0, y)
PAR_FER
ACC_OUV
(MOT_CLE, 112)
(IDENT, 120, 0, x)
PLUS
(IDENT, 121, 0, y)
SEP
ACC_FER
(IDENT, 62979, 0, int)
(IDENT, 19273, 0, main)
PAR_OUV
PAR_FER
ACC_OUV
(MOT_CLE, 173)
(IDENT, 36821, 0, variablelocal)
AFFEC
(CONST_INT, 6)
SEP
(MOT_CLE, 173)
(IDENT, 64494, 0, variableresultat)
AFFEC
(IDENT, 64881, 0, somme)
PAR_OUV
(IDENT, 36821, 0, variablelocal)
VIRG
(IDENT, 45542, 0, macro)
PAR_FER
SEP
ACC_FER
========== Table des identificateurs ==========
[x] Hash: 120

[y] Hash: 121

[main] Hash: 19273

[variablelocal] Hash: 36821

[macro] Hash: 45542

[int] Hash: 62979

[variableresultat] Hash: 64494

[somme] Hash: 64881

======= Fin Table des identificateurs ==========

Process returned 0 (0x0)   execution time : 0.108 s
Press any key to continue.
```

Partie Syntaxique

Démonstration de la transformation LL(1)

R1) <programme> : <liste-declarations> <liste-fonctions>

Ambiguité :

Exemple d'ambiguité : int a, b ;

int c[10] ;

int maFonction(int d)

Justification : Le principe LL1 ne permet de lire qu'un seul lexème à la fois ainsi en lisant int maFonction le compilateur n'est pas en mesure de déterminer si on est toujours dans une liste déclarations ou dans une liste des fonctions

En intégrant struct dans la grammaire améliorée et en remplaçant liste fonction et liste déclaration par leurs valeurs, on aboutit à :

<programme> : <liste-declaration><liste-fonctions>

Programme après remplacer liste fonction et liste déclaration par leurs valeurs devient :

<programme> :<declaration><liste-declaration> <liste-fonctions> | <liste-fonctions>

<programme> : <declaration><liste-declration> <fonction> <liste-fonction prime> | <fonction> <liste-fonctions prime>

On continue de remplacer les non terminaux pour pouvoir factoriser à la fin par int identificateur et void identificateur

On aboutit alors :

R11 : <programme> : int identificateur <programme'> | extern <programme''> | void identificateur (<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>|<bloc-struct><programme>

R12 : <programme'> : <declarateur prime> <liste declarateurprime> ; <programme>| (<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>

R13 : <programme''> : void identificateur (<liste-params>) ; <liste-fonctions prime> | int identificateur (<liste-params>) ; <liste-fonctions prime>

R 11 : on vérifie les premiers des productions :

Premier(int identificateur <programme'>) = { int }

Premier (extern <programme''>) = { extern }

Premier (void identificateur (<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>) = { void }

Premier (<bloc-struct><programme>) = { struct }

Les premiers des productions sont disjoints donc la règle R 11 est LL1

R 12 : on vérifie les premiers des productions :

Premier(<declarateur prime> <liste declarateurprime> ; <programme>) = {[, ' ,' , ;}

Premier((<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>

) = { (}

Les premiers des productions sont disjoints donc la règle R 12 est LL1

R13 :

Premier(void identificateur (<liste-params>) ; <liste-fonctions prime>) = {void}

Premier(int identificateur (<liste-params>) ; <liste-fonctions prime>) = { int}

Les premiers sont disjoints la règle R13 est LL1

R2) <liste-declarations> : <liste-declarations> <declaration> | Ɛ

Cette règle n'est pas LL1 elle présente une récursivité à gauche

Éliminer la récursivité :

R21 <liste-declarations> : <liste-declarations prime>

R22 <liste-declarations prime> : <declaration> <liste-declarations prime> | Ɛ

Pour optimiser la grammaire on élimine la première règle :

R2 : <liste-declarations> : <declaration> <liste-declarations> | Ɛ

R2 présente une production annulable, on vérifie alors le premier et suivant de liste-declarations

Pr(<liste-declarations>) = { int }

Suivant(<liste-declarations>) = { }, for, while, if, return, {, IDENT }

Donc l'intersection premier et suivant est vide, d'où la règle est LL1.

R3) <liste-fonctions> : <liste-fonctions> <fonction> | Ɛ

Cette règle implique qu'on pourrait avoir aucune fonction dans le programme, nous avons choisit d'avoir au moins une fonction en changeant la règle à :

R3 : <liste-fonctions> : <liste-fonctions> <fonction> | <fonction>

Cette règle n'est pas LL1 elle présente une récursivité à gauche

Éliminer la récursivité :

R 31 : <liste-fonctions> : <fonction> <fliste-fonctions prime>

R 32 : <liste-fonctions prime> : <fonction> <liste-fonctions prime> | Ɛ

R 31 : est LL1 car on a un seul choix

R 32 : admet une production annulable

Premier (<liste-fonctions prime>) = { Void ,extern,int, Ɛ}

Suivant(<liste-fonctions prime>) = {$}

Donc l'intersection du premier et suivant est vide, d'où la règle R 32 est LL1

R4 : <declaration> : int <liste-declarateurs> ; | struct identificateur < liste-declarateurStruct> ; | <struct>

On vérifie les premiers des productions

Premier(int <liste-declarateurs> ;) = { int}

Premier(struct identificateur < liste-declarateurStruct> ; | <struct>) = { struct}

L'intersection est donc vide, d'où la règle est LL1

R5) <liste-declarateurs> : <liste-declarateurs> , <declarateur> | <declarateur>

Cette règle présente une récursivité à gauche donc n'est pas LL1

Eliminer la récursivité à gauche :

R51 :<liste-declarateurs> : <declarateur> <liste-declarateur prime>

R52 :<liste-declarateurs prime> : , <declarateur> <liste-declarateurs prime> | Ɛ

R51 : Ok , car un seul choix

R52 : Admet une production annulable, on vérifier le premier et le suivant de la règle

Premier(<liste-declarateurs prime >) = { ‘,’, Ɛ }

Suivant(<liste-declarateurs prime>) = {;}

L’intersection du premier et suivant est vide, donc la règle est LL1

R6 : <declarateur> : identificateur | identificateur [constante] | identificateur . identificteur

La règle doit être factorisée

R61 : <declarateur> : identificateur <declarateur prime>

R62 : <declarateur prime> : [constante] | . identificateur | Ɛ

La règle R61 est LL1 car on a un seul choix

La règle R62 admet une production annulable

Premier(<declarateur-prime>) = { [, Ɛ }

Suivant(<declarateur – prime>) = { ‘,’, ; }

L’intersection est vide, donc la règle est LL1

R7 : < liste-declarateurStruct> : identificateur, < liste-declarateurStruct> | epsilon

Cette règle contient une production annulable, vérifions les premiers et les suivants :

Premier(< liste-declarateurStruct>) = { identificateur, epsilon}

Suivant(< liste-declarateurStruct>) = { ; }

L’intersection des premiers et suivant est vide donc la règle est LL1.

8 -

R8 : <fonction> : <type> identificateur (<liste-params>) {<liste-declarations>} <liste-instructions>} | extern <type> identificateur (<liste-params>) ;

On vérifie les premiers des production :

Premier(<type>) = { void, int}

Premier(extern) = {extern}

L'intersection des premiers est vide donc la règle est LL1

R9 : <type> : void | int

La règle est LL1 vu que les productions sont des terminaux

10 –

R10 : <liste-params> : <liste-params> , <param> | Ɛ

La règle présente une anomalie puisque une telle grammaire n'acceptera pas cette écriture :

Int calcul(int x, int y)

Mais plutôt celle-là :

Int calcul(,int x, int y)

On a remédier à cette anomalie par ces deux règles :

R 101 : <liste-params> : <param><liste-params prime>| Ɛ

R 102 : <liste-params prime> : , <param> <liste-params prime> | Ɛ

On vérifie pour R 101 le premier et suivant vu que la règle admet une production annulable

Premier(<liste-params>) = { int , Ɛ }

Suivant(<lite-params>} = {) }

L'intersection des suivant et premier est vide donc la règle est LL1

On vérifie pour R 102 le premier et suivant vu que la règle admet une production annulable

Premier(<liste-params prime>) = { ',', Ɛ }

Suivant(<lite-params>} = {) }

L'intersection des suivant et premier est vide donc la règle est LL1

R11) <param> : int identificateur

La règle est LL1 car elle admet un seul choix.

R12) <liste-instructions> : <liste-instruction> <instruction> | Ɛ

La règle admet une récursivité à gauche.

On élimine la récursivité à gauche :

R121 :<liste-instruction> : <liste-instructions prime>

R121 : <liste-instruction prime> : <instruction> <liste-instructions prime> | Ɛ

On optimise la grammaire en éliminant la règle R121

R11 : <liste-instruction> : <instruction> <liste-instruction> | Ɛ

On vérifie le premier et suivant de la règle vu qu'elle admet une production annulable

Premier(<liste-instruction>) = {[, =}

Suivant (<liste-instruction>) = { For, while, if, return, {, IDENT }

L'intersection est vide donc la règle est LL1

<instruction> : <iteration> | <selection> | <saut>| <affectation> ; | <bloc> | <appel>

On vérifie les premiers de chaque non terminal

Premier(<iteration>) = { For, while}

Premier(<selection>) = { If }

Premier(<saut>) = { Return}

Premier(<affectation>) = {identificateur}

Premier(<bloc>) = { '{'}

Premier(<appel>) = { identificateur}

Intersection de <appel> et de <affectation> n'est pas vide, on remplace alors les non-terminaux

On obtient alors :

R131 : <instruction> : <iteration> | <selection> | <saut>| bloc | <variable > = <expression> | identificateur (<liste-expressions>) ;

On remplace variable :

<instruction> : <iteration> | <selection> | <saut>| bloc | identificateur = <expression> | identificateur [<expressions>] = <expression> | identificateur (<liste-expressions>) ;

On factorise par identificateur :

R131 : <instruction> : : <iteration> | <selection> | <saut>| bloc | identificateur <instruction prime>

R132 : <instruction prime> : [<expression>] = <expression> | (<liste-expression>) ;| = <expression>

On vérifie si R121 est LL1 :

Premier(<iteration>) = { For, while}

Premier(<selection>) = { If }

Premier(<saut>) = { Return}

Premier(<bloc>) = { '{'}

Premier(identificateur) = {identificateur}

L'intersection des production est vide donc R121 est vide

On vérifie R122 est LL1 :

Premier([<expression>]) = { [}

Premier ((<liste-expression>) ;) = { (}

Premier { = <expression> } = { = }

Les premiers sont disjoints donc la règle est LL1

R14) <iteration> : for (<affectation> ; <condition> ; <affectation>) <instruction> | while (<condition>) <instruction>

Premier (for (<affectation> ; <condition> ; <affectation>) <instruction>) = { for}

Premier (while (<condition>) <instruction>) = { while}

Donc les premiers sont disjoints alors la règle est LL1

R15) <selection> : if (<condition>) <instruction> | if (<condition>) <instruction> else <instruction>

Factorisation par if:

R 151: <selection> : if (<condition>) <instruction> <selection prime>

R 152: <selection prime> : else <instruction> | Ɛ

R 151 est LL1 car un seul choix

R 152 admet une production annulable :

Premier(<selection prime>) = { Else , Ɛ }

Suivant (<selection prime>) = { For, while, if, return, {, IDENT }

Premier et suivant sont disjoints donc la règle R 142 est LL1

R16) <saut> : return ; | return <expression> ;

Factorisation par return

R 161 : <saut> : return <saut prime>

R 162 : <saut prime> : ; | <expression> ;

R 161 est LL1 car un seul choix

R 162 on vérifie les premiers :

Premier (;) = { ; }

Premier (<expression> ;) = { (, - , IDENT , Const }

Donc l'intersection est vide donc la règle R 152 est LL1

R17) <affectation> : <variable> = <expression>

Cette règle n'est plus utilisée puisqu'elle a été remplacée par sa valeur dans toutes les règles où elle est mentionnée

R18) <bloc> : { <liste-instructions> }

Règle est LL1 car on a choix unique

R19) <appel> : identificateur (<liste-expressions>) ;

Cette règle n'est plus utilisée puisqu'elle a été remplacée par sa valeur dans toutes les règles où elle est mentionnée

R20) <variable > : identificateur | identificateur [<expression>] | identificateur . identificateur

Factorisation par identificateur :

R 201 : <variable> : identificateur <variable prime>

R 202 : <variable prime> : [<expression>] | . identificateur | Ɛ

R 201 est LL1 car un seul choix

R 202 on vérifie le premier et le suivant :

Premier(<variable prime>) = { [, . , Ɛ}

Suivant(<variable prime>) = { + , - ,* , / , << , >> , & , | , ',' }

Premier et suivant sont disjoints donc R 192 est LL1

R21) <expression> = (<expression>) | <expression> <binary-op><expression> | - <expression> | <variable> | identificateur (<liste-expression>)

On élimine la récursivité à gauche :

R211 : <expression> : (<expression>) <expression prime> | - <expression> <expression prime> | constante <expression prime> | identificateur <expression seconde>

R212 : <expression prime> : <binary-op> <expression> <expression prime> | Ɛ

R213 : <expression seconde> : <variable prime><expression prime> | (<liste-expressions>)

<expression prime>

Vérifions que la règle R201 est LL1 :

Premier ((<expression>)) <expression prime>) = { (}

Premier (constante <expression prime>) = { Const }

Premier (- <expression> <expression prime>) = { - }

Premier(identificateur <expression seconde>) = { identificateur }

Les premiers sont disjoints donc la règle est LL1

R22): <listes-expressions> : <listes-expressions> , <expression> | epsilon

La règle presente une anomalie puisque une telle grammaire n'acceptera pas cette écriture :

Int calcul(4+4,9*2)

Mais plutôt celle-là :

Int calcul(,4+4,9*2)

On a remédier à cette anomalie par ces deux règles :

R 221 : <liste-expressions> : <expression><liste-expressionsprime> | Ɛ

R 222 : <liste-expressionsprime> : , <expression><liste-expressionsprime> | Ɛ

R 221 : admet une production annulable

On vérifie le premier et le suivant

Premier(<liste-expressions>) = { (, - , IDENT , Const , Ɛ}

Suivant (<liste-expressions>) = {) }

Les premiers et suivants sont disjoints donc R 211 est LL1

R212 : admet une production annulable

On vérifie le premier et le suivant

Premier(<liste-expressionsprime>) = { ',' , Ɛ}

Suivant (<liste-expressionsprime>) = {) }

Les premiers et suivants sont disjoints donc R 212 est LL1

R23) <condition> : ! (<condition>) | <condition> <binary-rel> <condition> | (<condition>) | <expression><binary-comp><expression>

Éliminer la récursivité à gauche :

R 231: <condition> : ! (<condition>) < condition prime> | (<condition>) <condition prime> | <expression> <binary-comp> <expression> <condition prime>

R 232: <condition prime> : <binary-rel> <condition> <condition prime> | Ɛ

Vérifions la règle R221 :

Premier (! (<condition>) < condition prime>) = { ! }

Premier ((<condition>) <condition prime>) = { (}

Premier (<expression> <binary-comp> <expression> <condition prime>) = { (, - , IDENT , Const }

R24) <struct> : struct identificateur { <liste-attributs>} ;

La règle est LL1 car on a un choix unique.

R25) <liste-attributs> : int <declarateur> ; <liste-attributs> | int <declarateur> ;

Factorisation :

R 251 : <liste-attributs> : int <declarateur> ; < liste-attribut prime>

R 252 : <liste-attributs prime> : int <declarateur> ; < liste-attribut prime> | Ɛ

R 251 : La règle est LL1 car il existe un seul choix

R 252 : On a ajouter int <declarateur> de nouveau pour éviter une règle récursive

On vérifie qu'elle est LL1 en vérifiant les premiers et suivants :

Premier(<liste-attributs prime>) = { int, Ɛ }

Suivant (<liste-attributs prime>) = { } }

Premier et suivant disjoints donc règle R242 est LL1.

R 26) <bloc-struct> : <bloc-struct> <declaration'> | struct identificateur

La règle n'est pas LL1 vu qu'elle présente une récursivité à gauche

On élimine la récursivité :

R 261 :<bloc-struct> : struct identificateur<bloc-struct'> | Ɛ

R 262 :<bloc-struct'> : <declaration'> <bloc-struct>

La règle R 261 contient une production annullable ont vérifie les premiers et les suivants:

Premier(<bloc-struct>) = { struct, epsilon}

Suivant(<bloc-struct>) = { void, int, extern}

L'intersection est vide donc la règle est LL1

La règle R 262 est LL1 puisqu'elle comporte un seul choix.

R27) <binary-op> : + | - | * | / | << | >> | & | | |

Règle lexicale.

R28) <binary-rel> : && | ||

Règle lexicale.

R29) <binary-comp> : < | > | >= | <= | == | !=

Règle lexicale.

Transformation à une grammaire LL(1)

La grammaire que nous possédons actuellement n'est pas **LL(1).** En effet, de nombreuses règles de productions le montrent clairement.

Récursivité à gauche :

<liste-declaration> : <liste-declaration><declaration> | epsilon

<liste-fonctions> : <liste-fonctions> <fonction> | epsilon

<liste-declarateurs> : <liste-declarateurs> , <declarateur> | <declarateur>

Il est donc indispensable de rendre la grammaire **LL(1)** afin de pouvoir continuer et en tirer l'arbre syntaxique. Pour ce, nous allons éliminer les récursivités à gauche, factoriser à gauche, et chercher les premiers et les suivants des non-terminaux.

<programme> : int identificateur <programme'> | extern <programme''> | void identificateur (<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>|<bloc-struct><programme>

<programme'> : <declarateur prime> <liste declarateurprime> ; <programme>| (<liste-params>) { <liste-declarations> <liste-instructions> } <liste-fonctions prime>

<programme''> : void identificateur (<liste-params>) ; <liste-fonctions prime> | int identificateur (<liste-params>) ; <liste-fonctions prime>

<bloc-struct> : struct identificateur<bloc-struct'> | Ɛ

<bloc-struct'> <declaration'> <bloc-struct>

<liste-declarations> : <declaration> <liste declarations> | Ɛ

<liste-fonctions> : <fonction><liste-fonctions prime>

<liste-fonctions prime> : <fonction> <liste-fonctions prime> | Ɛ

<declaration> : int <liste-declarateurs> ;| struct identificateur<declaration'>

<declaration'> : identificateur < liste-declarateurs-struct> ; | { <liste-attributs> };

< liste-declarateurStruct> :, identificateur < liste-declarateurStruct> | Ɛ

<liste-declarateurs> : <declarateur> <liste-declarateurs prime>

<liste-declarateurs prime> : , <declarateur> <liste-declarateurs prime> | Ɛ

<declarateur> : identificateur <declarateur prime>

<declarateur prime> : [constante] | Ɛ

<fonction> : <type> identificateur (<liste-params>) { <liste-declarations> <liste-instructions> } |

extern <type> identificateur (<liste-params>) ;

<type> : void | int

<liste-params> : <param><liste-paramsprime>

<liste-paramsprime> : , <param><liste-paramsprime> | Ɛ

<param> : int identificateur

<liste-instructions> : <instruction> <liste-instructions> | Ɛ

<instruction> : <iteration> | <selection> | <saut> | <bloc> | identificateur <instruction prime>

<instruction prime> : [<expression>] = <expression> | (<liste-expressions>) ; | = <expression>

<iteration> : for (<affectation> ; <condition> ; <affectation>) <instruction> | while (<condition>) <instruction>

<selection> : if (<condition>) <instruction> <selection prime>

<selection prime> : else <instruction> | Ɛ

<saut> : return <saut prime>

<saut prime> : <expression> ; | ;

<bloc> : { <liste-instructions>}

<variabe prime> : [<expression>] | . identificateur | Ɛ

<expression> : (<expression>) <expression prime> | - <expression> <expression prime> | |constante <expression prime> | identificateur <expression seconde>

<expression seconde> : <variable prime><expression prime> | (<liste-expressions>)

<expression prime>

<expression prime> : <binary-op> <expression> <expression prime> | Ɛ

<liste-expressions> : <expression><liste-expressionsprime> | Ɛ

<liste-expressionsprime> : , <expression><liste-expressionsprime> | Ɛ

<condition> : ! (<condition>) < condition prime> | (<condition>) <condition prime> | <expression> <binary-comp> <expression> <condition prime>

<condition prime> : <binary-rel> <condition> <condition prime> | Ɛ

<struct> : struct identificateur { <liste-attributs> };

<liste-attributs> : int <declarateur> ; <liste-attributs prime>

<liste-attributs prime> : int <declarateur> ; <liste attributs prime> | Ɛ

<binary-op> : + | - | * | / | << | >> | & | | |

<binary-rel> : && | ||

<binary–comp> : < | > | >= | <= | == | !=

Table des premiers et suivants

Non terminal	Premier	Suivant	LL1
<programme>	Struct,int,extern,void	$	Ok : un seul choix
<programme'>	Int,extern,void	$	Ok : intersection vide
<programme ''>	[, ; , ',' , (	$	Ok : intersection vide
<programme'''>	Void,int	$	Ok : intersection vide
<bloc-struct>	Struct, Ɛ	Int,extern,void	Ok : inter sui et pre vide
<liste-declaration>	Int	},for,while,if,return,{,IDENT	Ok : choix unique
<liste-declarations prime>	','	},for,while,if,return,{,IDENT	Ok : inter sui et pre vide
<liste-fonctions>	Void ,extern,int	$	Ok : choix unique
<liste-fonctions prime>	Void ,extern,int, Ɛ	$	Ok : inter sui et pre vide
<declaration>	Int	',',},for,while,if,return,{,IDENT	Ok : choix unique
<liste-declarateurs>	IDENT	;	Ok : choix unique
<liste-declarateurs prime>	',', Ɛ	;	Ok : inter pre et suivant vide
<declarateur>	IDENT	',', ;	Ok : choix unique
<declarateur prime>	[, Ɛ	',', ;	Ok : inter sui et pre est vide
<fonction>	Void, int, extern	Void, int, extern	Ok : inter premier vide
<liste-params>	',' , Ɛ	)	Ok : choix Unique
<liste-params prime>	',' , Ɛ	)	Ok : inter suivant et pre vide
<param>	Int	',' ,)	Ok : choix unique
<liste-instructions>	For, while, if, return, {, IDENT, Ɛ	}	Ok : choix unique
<liste-instructions prime>	For, while, if, return, {, IDENT, Ɛ	}	Ok : inter suivant et pre vide
<instruction>	For, while, if, return, {, IDENT	For, while, if, return, {, IDENT	Ok : inter premiers vide
<instruction prime>	[, =	For, while, if, return, {, IDENT	Ok : inter des premiers vide
<iteration>	For, while	For, while, if, return, {, IDENT	Ok : inter des premiers vide
<selection>	If	For, while, if, return, {, IDENT	Ok : choix unique
<selection prime>	Else , Ɛ	For, while, if, return, {, IDENT	Ok : inter suivant pre vide
<saut>	Return	For, while, if, return, {, IDENT	Ok: Choix unique
<saut prime>	(,const,-,IDENT, ;	For, while, if, return, {, IDENT	Ok: inter des premiers vide
<bloc>	{	For, while, if, return, {, IDENT	Ok :Choix Unique
<variable prime >	[, . , Ɛ	+ , - ,* , / , << , >> , & , | , ','	Ok : inter des premiers et des suivants et vide
<expression>	(, - , IDENT , Const	] ,) , + , - ,* , / , << , >> , & , | , ',' , ! , IDENT , Const , (, && , ||	Ok : inter premiers vide
<expression seconde>	[, . , Ɛ , (	] ,) , + , - ,* , / , << , >> , & , | , ',' , ! , IDENT , Const , (, && , ||	Ok : premiers disjoints
<expression prime>	+ , - ,* , / , << , >> , & , | , Ɛ		

Difficultés rencontrées (Syntaxique)

- L'intégration de l'amélioration struct a engendré plusieurs problèmes dans la grammaire en causant des ambiguïtés.

1) Ambiguïté au début du programme :

Lors de la déclaration des variables et des fonctions, notre compilateur n'a pas les moyens de savoir si la déclaration qu'il est en train de lire est celle d'une variable simple ou d'un tableau ou une fonction. Pour ce il a fallu changer un peu la première règle en remplaçant et en factorisant jusqu'à ce que l'appel de la fonction suivante ne dépend pas des deux premiers lexèmes qui causaient l'ambigüité : int identificateur.

2) Ambiguïté dans expression :

- L'intégration de struct au sein des fonctions : comment déclarer un struct dans la fonction
- L'initialisation des attributs d'un struct dans la fonction

Ces difficultés qui surviennent à chaque fois, nous ont obligés à changer la concéption et la grammaire plusieurs fois en changeant aussi le code à maintes reprises. Ces changements fréquents ont causé un retard en ce qui concerne l'avancement du projet et le passage à l'étape suivant qui est la sémantique.

Annexe

Lexical.h

```
1.  #ifndef LEXICAL_H
2.  #define LEXICAL_H
3.
4.  #include <bits/stdc++.h>
5.  #include "Structures.h"
6.
7.  using namespace std;
8.
9.  class Lexical
10. {
11.   private:
12.     string tableHachageMC[250];
13.     vector<string> tableHachageID[100000];
14.     char carEnAvance;
15.
16.   public:
17.
18.     Lexical(); //Constructeur de la classe
19.     TUniteLexicale uniteSuivante(); //Utilisée par le lexical pour savoir l'unité lexicale suiv
    ante à lire
20.     char lireCar(); //Permet de lire un caractère
21.     void enMiniscule(string &s); //Transforme la chaine fournie en miniscule, car en C-
    - les identificateurs ne sont pas sensibles à la casse.
22.     void ajouterCar(string &lexeme); //Ajouter un caractère au léxème
23.     bool estBlanc(char c); //Traite le cas où un caractère est blanc
24.     bool estLettre(char c); //Traite le cas où le caractère est une lettre
25.     bool estChiffre(char c); //Traite le cas où le caractère est un chiffre
26.     int convertirChaineValeurEntier(string lexeme); //Convertit le lexème à sa valeur en entier

27.     int rechercheDansMotCle(string lexeme); //Convertit le lexème à sa valeur en entier
28.     int rechercheAjoutDansListeIdentificateurs(string lexeme);//Recherche le lexème donnée dans
     la liste des ident et l'ajoute s'il n'y existe pas
29.     int hashMotCle(string s); //Hashe le mot clé fourni
30.     int hashIdent(string s); // Hashe l'ident fourni
31.     void afficheTableHachageID();
    //Affiche la table de hachage des identificateurs avec un formattage agréable
32.     string getIdentificateur(int pos, int posConflit);
33.
34.
35. };
36.
37. #endif // LEXICAL_H
```

Constantes.h

```
#ifndef CONSTANTES_H_INCLUDED
#define CONSTANTES_H_INCLUDED

#include <bits/stdc++.h>
using namespace std;

string listeMotsCle[] =
{
    "if",
    "then",
    "else",
    "while",
    "for",
    "extern",
    "int",
    "void",
    "return",
    "struct"
};

#endif // CONSTANTES_H_INCLUDED
```

Structures.h

```
#ifndef STRUCTURES_H_INCLUDED
#define STRUCTURES_H_INCLUDED

typedef enum
{
    IDENT,
    CONST_INT,
    MOT_CLE,
    VIRG,
    NON,
    PLUS,
    MOINS,
    MULT,
    DIV,
    DEC_DROITE,
    DEC_GAUCHE,
    BIN_ET,
    BIN_OUT,
    ET,
    OU,
    INF,
    SUP,
    INFEGAL,
    SUPEGAL,
    EGAL,
    DIFF,
    AFFEC,
    SEP,
    PAR_OUV,
    PAR_FER,
    CRO_OUV,
    CRO_FER,
    ACC_OUV,
    ACC_FER,
    DEB_COM,
    FIN_COM,
    POINT,
    ERREUR,
    ENDOF
} TUnite;

typedef struct TUniteLexicale
{
    TUnite UL;
    int attribut;
    int positionConflit;
} TUniteLexicale;

#endif // STRUCTURES_H_INCLUDED
```

Lexical.cpp

```cpp
#include "Lexical.h"
#include "Constantes.h"

using namespace std;

// Le constructeur hashe les mots clés
Lexical ::Lexical()
{
    for (int i = 0; i < 10; i++)
    {
        int pos = hashMotCle(listeMotsCle[i]);
        tableHachageMC[pos] = listeMotsCle[i];
    }
    carEnAvance = lireCar();
}

char Lexical::lireCar()
{
    return getc(stdin);
}

void Lexical::ajouterCar(string &lexeme)
{
    lexeme += carEnAvance;
}

bool Lexical::estBlanc(char c)
{
    return (c == ' ') || (c == '\t') || (c == '\n') || (c == '\v');
}

bool Lexical::estLettre(char c)
{
    return (c >= 'A' && c <= 'Z') || (c >= 'a' && c <= 'z');
}

bool Lexical::estChiffre(char c)
{
    return isdigit(c);
}

// Retourne l'unité lexicale suivante
TUniteLexicale Lexical::uniteSuivante()
{
    while (estBlanc(carEnAvance))
    {
        carEnAvance = lireCar();
    }

    /* Dès que notre caractère est différent de blanc, on étudie les cas possibles.
    On retourne le léxème correspondant et l'attribut dans chaque cas en suivant l'automate construit.
    */

    switch (carEnAvance)
    {
    case ',':
    {
        TUniteLexicale lexeme;
        lexeme.UL = VIRG;
        carEnAvance = lireCar();
        return (lexeme);
    }
    case '+':
    {
        TUniteLexicale lexeme;
```

```
66.         lexeme.UL = PLUS;
67.         carEnAvance = lireCar();
68.         return (lexeme);
69.     }
70.     case '-':
71.     {
72.         TUniteLexicale lexeme;
73.         lexeme.UL = MOINS;
74.         carEnAvance = lireCar();
75.         return (lexeme);
76.     }
77.     case ';':
78.     {
79.         TUniteLexicale lexeme;
80.         lexeme.UL = SEP;
81.         carEnAvance = lireCar();
82.         return (lexeme);
83.     }
84.     case '(':
85.     {
86.         TUniteLexicale lexeme;
87.         lexeme.UL = PAR_OUV;
88.         carEnAvance = lireCar();
89.         return (lexeme);
90.     }
91.     case ')':
92.     {
93.         TUniteLexicale lexeme;
94.         lexeme.UL = PAR_FER;
95.         carEnAvance = lireCar();
96.         return (lexeme);
97.     }
98.     case '[':
99.     {
100.             TUniteLexicale lexeme;
101.             lexeme.UL = CRO_OUV;
102.             carEnAvance = lireCar();
103.             return (lexeme);
104.         }
105.         case ']':
106.         {
107.             TUniteLexicale lexeme;
108.             lexeme.UL = CRO_FER;
109.             carEnAvance = lireCar();
110.             return (lexeme);
111.         }
112.         case '{':
113.         {
114.             TUniteLexicale lexeme;
115.             lexeme.UL = ACC_OUV;
116.             carEnAvance = lireCar();
117.             return (lexeme);
118.         }
119.         case '}':
120.         {
121.             TUniteLexicale lexeme;
122.             lexeme.UL = ACC_FER;
123.             carEnAvance = lireCar();
124.             return (lexeme);
125.         }
126.         case '!':
127.         {
128.             carEnAvance = lireCar();
129.             if (carEnAvance == '=')
130.             {
131.                 TUniteLexicale lexeme;
132.                 lexeme.UL = DIFF;
133.                 carEnAvance = lireCar();
134.                 return (lexeme);
```

```
135.                }
136.                else
137.                {
138.                    TUniteLexicale lexeme;
139.                    lexeme.UL = NON;
140.
141.                    return (lexeme);
142.                }
143.            }
144.            case '/':
145.            {
146.                carEnAvance = lireCar();
147.                if (carEnAvance == '*')
148.                {
149.                    TUniteLexicale lexeme;
150.                    lexeme.UL = DEB_COM;
151.                    carEnAvance = lireCar();
152.                    return (lexeme);
153.                }
154.                else
155.                {
156.                    TUniteLexicale lexeme;
157.                    lexeme.UL = DIV;
158.
159.                    return (lexeme);
160.                }
161.            }
162.            case '*':
163.            {
164.                carEnAvance = lireCar();
165.                if (carEnAvance == '/')
166.                {
167.                    TUniteLexicale lexeme;
168.                    lexeme.UL = FIN_COM;
169.                    carEnAvance = lireCar();
170.                    return (lexeme);
171.                }
172.                else
173.                {
174.                    TUniteLexicale lexeme;
175.                    lexeme.UL = MULT;
176.
177.                    return (lexeme);
178.                }
179.            }
180.            case '&':
181.            {
182.                carEnAvance = lireCar();
183.                if (carEnAvance == '&')
184.                {
185.                    TUniteLexicale lexeme;
186.                    lexeme.UL = ET;
187.                    carEnAvance = lireCar();
188.                    return (lexeme);
189.                }
190.                else
191.                {
192.                    TUniteLexicale lexeme;
193.                    lexeme.UL = BIN_ET;
194.
195.                    return (lexeme);
196.                }
197.            }
198.            case '|':
199.            {
200.                carEnAvance = lireCar();
201.                if (carEnAvance == '|')
202.                {
203.                    TUniteLexicale lexeme;
```

```
204.                     lexeme.UL = OU;
205.                     carEnAvance = lireCar();
206.                     return (lexeme);
207.                 }
208.                 else
209.                 {
210.                     TUniteLexicale lexeme;
211.                     lexeme.UL = BIN_OUT;
212.
213.                     return (lexeme);
214.                 }
215.             }
216.             case '=':
217.             {
218.                 carEnAvance = lireCar();
219.                 if (carEnAvance == '=')
220.                 {
221.                     TUniteLexicale lexeme;
222.                     lexeme.UL = EGAL;
223.                     carEnAvance = lireCar();
224.                     return (lexeme);
225.                 }
226.                 else
227.                 {
228.                     TUniteLexicale lexeme;
229.                     lexeme.UL = AFFEC;
230.
231.                     return (lexeme);
232.                 }
233.             }
234.             case '<':
235.             {
236.                 carEnAvance = lireCar();
237.                 if (carEnAvance == '<')
238.                 {
239.                     TUniteLexicale lexeme;
240.                     lexeme.UL = DEC_DROITE;
241.                     carEnAvance = lireCar();
242.                     return (lexeme);
243.                 }
244.                 else if (carEnAvance == '=')
245.                 {
246.                     TUniteLexicale lexeme;
247.                     lexeme.UL = INFEGAL;
248.                     carEnAvance = lireCar();
249.                     return (lexeme);
250.                 }
251.                 else
252.                 {
253.                     TUniteLexicale lexeme;
254.                     lexeme.UL = INF;
255.
256.                     return (lexeme);
257.                 }
258.             }
259.             case '>':
260.             {
261.                 carEnAvance = lireCar();
262.                 if (carEnAvance == '>')
263.                 {
264.                     TUniteLexicale lexeme;
265.                     lexeme.UL = DEC_GAUCHE;
266.                     carEnAvance = lireCar();
267.                     return (lexeme);
268.                 }
269.                 else if (carEnAvance == '=')
270.                 {
271.                     TUniteLexicale lexeme;
272.                     lexeme.UL = SUPEGAL;
```

```
273.                     carEnAvance = lireCar();
274.                     return (lexeme);
275.                 }
276.                 else
277.                 {
278.                     TUniteLexicale lexeme;
279.                     lexeme.UL = SUP;
280.                     return (lexeme);
281.                 }
282.             }
283.             case '_':
284.             {
285.                 carEnAvance = lireCar();
286.                 TUniteLexicale lexeme;
287.                 lexeme.UL = ERREUR;
288.                 lexeme.attribut = 1;
289.                 return (lexeme);
290.             }
291.
292.             case '.':
293.             {
294.                 carEnAvance = lireCar();
295.                 TUniteLexicale lexeme;
296.                 lexeme.UL = POINT;
297.                 return (lexeme);
298.             }
299.
300.             case EOF:
301.             {
302.                 TUniteLexicale lexeme;
303.                 lexeme.UL = ENDOF;
304.                 return (lexeme);
305.             }
306.             default:
307.             {
308.                 // Cas des chiffres et des lettres
309.                 string lexemeLu = "";
310.                 if (estChiffre(carEnAvance))
311.                 {
312.                     while (estChiffre(carEnAvance))
313.                     {
314.                         ajouterCar(lexemeLu);
315.                         carEnAvance = lireCar();
316.                     }
317.                     TUniteLexicale lexeme;
318.                     lexeme.UL = CONST_INT;
319.                     lexeme.attribut = convertirChaineValeurEntier(lexemeLu);
320.                     return (lexeme);
321.                 }
322.                 else if (estLettre(carEnAvance))
323.                 {
324.                     while (estChiffre(carEnAvance) || estLettre(carEnAvance) || carEn
    Avance == '_')
325.                     {
326.                         ajouterCar(lexemeLu);
327.                         carEnAvance = lireCar();
328.                     }
329.
330.                     //Si le mot clé existe on retourne l'unité lexicale et son attrib
    ut.
331.                     if (rechercheDansMotCle(lexemeLu) != -1)
332.                     {
333.
334.                         TUniteLexicale lexeme;
335.                         lexeme.UL = MOT_CLE;
336.                         lexeme.attribut = rechercheDansMotCle(lexemeLu);
337.                         return (lexeme);
338.                     }
339.                     else
```

```
340.                    {
341.                        TUniteLexicale lexeme;
342.                        lexeme.UL = IDENT;
343.                        lexeme.attribut = rechercheAjoutDansListeIdentificateurs(lexe
    meLu);
344.                        lexeme.positionConflit = tableHachageID[lexeme.attribut].size
    () - 1;
345.                        return (lexeme);
346.                    }
347.                }
348.                else
349.                {
350.                    carEnAvance = lireCar();
351.                    TUniteLexicale lexeme;
352.                    lexeme.UL = ERREUR;
353.                    lexeme.attribut = 2;
354.                    return (lexeme);
355.                }
356.            }
357.            }
358.        }
359.
360.        int Lexical::convertirChaineValeurEntier(string lexeme)
361.        // Convertit la chaine fournie en entier en utilisant stoi.
362.        {
363.            return stoi(lexeme);
364.        }
365.
366.        int Lexical::hashMotCle(string s)
367.        {
368.            // Hashing simple des mots clés
369.            int val = 0;
370.            for (int i = 0; i < s.size(); i++)
371.            {
372.                val += s[i] * (i + 1);
373.            }
374.            return val % 250;
375.        }
376.
377.        int Lexical::hashIdent(string s)
378.        {
379.            int val = 0;
380.            int p = 1;
381.            for (int i = 0; i < s.size(); i++)
382.            {
383.                val = (val + s[i] * p) % 100000;
384.                p = (p * 37) % 100000;
385.            }
386.            return val % 100000;
387.        }
388.
389.        int Lexical::rechercheDansMotCle(string lexeme)
390.        {
391.            //Recherche le mot clé fourni dans la table des mots clés hashés
392.            int pos = hashMotCle(lexeme);
393.            if (tableHachageMC[pos] == lexeme)
394.                return pos;
395.            return -1;
396.        }
397.
398.        void Lexical::enMiniscule(string &s)
399.        {
400.            transform(s.begin(), s.end(), s.begin(), ::tolower);
401.        }
402.
403.        int Lexical::rechercheAjoutDansListeIdentificateurs(string lexeme)
404.        {
405.            //On transforme le léxème en miniscule avant de commencer les traitements
```

```
406.            enMiniscule(lexeme);
407.            int pos = hashIdent(lexeme);
408.            bool IDExistant = false;
409.            //On parcourt la table des ident, si l'ident existe on retourne sa positi
    on.
410.            for (int i = 0; i < tableHachageID[pos].size(); i++)
411.            {
412.                if (tableHachageID[pos][i] == lexeme)
413.                    IDExistant = true;
414.            }
415.            if (!IDExistant) // si l'ident n'existe pas on l'ajoute
416.            {
417.                tableHachageID[pos].push_back(lexeme);
418.
419.            }
420.            return pos;
421.        }
422.
423.        void Lexical::afficheTableHachageID()
424.        {
425.            cout << "========== Table des identificateurs ==========" << endl;
426.            for(int i = 0; i < 100000; i++)
427.            {
428.                if(tableHachageID[i].size() > 0)
429.                {
430.                    cout << "[";
431.                    for(int j = 0; j < tableHachageID[i].size(); j++)
432.                    {
433.                        // Si on est à l'élément l'avant dernier du tableau, on compl
    ète par une virgule.
434.                        if(j < tableHachageID[i].size() - 1)
435.                            cout << tableHachageID[i].at(j) << ", ";
436.                        else
437.                        {
438.                            cout << tableHachageID[i].at(j) << "] ";
439.                        }
440.                    }
441.                    cout << "Hash: " << i << endl;
442.                    cout << endl;
443.                }
444.
445.            }
446.            cout << "======= Fin Table des identificateurs ==========" << endl;
447.        }
448.
449.        string Lexical::getIdentificateur(int pos, int positionConflit)
450.        {
451.            return tableHachageID[pos][positionConflit];
452.        }
```

Syntaxique.h

```
#ifndef SYNTAXIQUE_H_INCLUDED
#define SYNTAXIQUE_H_INCLUDED

#include "Lexical.h"
#include <iostream>
#include <stdlib.h>
#include <string.h>
#include <stdio.h>

class Syntaxique {
    public:
        Syntaxique();
        bool affectation();
        bool programme();
        bool blocStruct();
        bool blocStructPrime();
        bool programmePrime();
        bool programmeSeconde();
        bool programmeTierce();
        bool listeParams();
        bool listeParamsPrime();
        bool declaration();
        bool declarationPrime();
        bool declarateurPrime();
        bool listeDeclarations();
        bool listeDeclarateurs();
        bool listeDeclarateursPrime();
        bool listeDeclarateursStruct();
        bool listeDeclarationsPrime();
        bool listeInstructions();
        bool instruction();
        bool listeInstructionsPrime();
        bool listeFonctions();
        bool listeFonctionsPrime();
        bool declarateur();
        bool fonction();
        bool parametre();
        bool instructionPrime();
        bool type();
        bool iteration();
        bool selection();
        bool selectionPrime();
        bool saut();
        bool listeExpressionsPrime();
        bool sautPrime();
        bool bloc();
        bool variable();
        bool variablePrime();
        bool expression();
        bool expressionPrime();
        bool expressionSeconde();
        bool expressionTierce();
        bool listeExpressions();
        bool condition();
        bool conditionPrime();
        bool structure();
        bool listeAttributs();
        bool listeAttributsPrime();
        bool binaryOp();
        bool binaryRel();
        bool binaryComp();

    private:
        TUniteLexicale motCourant;
        Lexical lex;
};
```

Syntaxique.cpp

```cpp
#include "Lexical.h"
#include "Syntaxique.h"
#include "string.h"

Syntaxique::Syntaxique() : lex()
{
    motCourant = lex.uniteSuivante();
}

bool Syntaxique::programme()
{
    return programmePrime();
}

bool Syntaxique::blocStruct()
{
    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("struct"))
    {
        motCourant = lex.uniteSuivante();
        if (motCourant.UL == IDENT)
        {
            motCourant = lex.uniteSuivante();
            return blocStructPrime();
        }
        else
        {
            cout << "Erreur: identificateur attendu dans blocStruct" << endl;
            return false;
        }
    }
    else if (motCourant.UL == MOT_CLE && (motCourant.attribut == lex.hashMotCle("int") ||
                                          motCourant.attribut == lex.hashMotCle("extern") ||
                                          motCourant.attribut == lex.hashMotCle("void")))
    {
        return true;
    }
    else
    {
        return false;
    }
}

bool Syntaxique::blocStructPrime()
{
    if (motCourant.UL == IDENT || motCourant.UL == ACC_OUV)
    {
        if (declarationPrime())
        {
            return blocStruct();
        }
        else
        {
            return false;
        }
    }
    else
    {
        cout << "Erreur: identificateur ou { non reconnue dans blocStruct" << endl;
        return false;
    }
}

bool Syntaxique::declarationPrime()
{
    if (motCourant.UL == IDENT)
    {
```

```
67.         motCourant = lex.uniteSuivante();
68.         if(listeDeclarateursStruct())
69.         {
70.             if(motCourant.UL == SEP)
71.             {
72.                 motCourant = lex.uniteSuivante();
73.                 return true;
74.             }
75.             else
76.             {
77.                 cout << "Erreur: ; attendu dans declarationPrime" << endl;
78.                 return false;
79.             }
80.         }
81.         else
82.         {
83.             return false;
84.         }
85.     }
86.     else if(motCourant.UL == ACC_OUV)
87.     {
88.         motCourant = lex.uniteSuivante();
89.         if(listeAttributs())
90.         {
91.             if(motCourant.UL == ACC_FER)
92.             {
93.                 motCourant = lex.uniteSuivante();
94.                 if(motCourant.UL == SEP)
95.                 {
96.                     motCourant = lex.uniteSuivante();
97.                     return true;
98.                 }
99.                 else
100.                {
101.                    cout << "Erreur: ; attendu dans declarationPrime" << endl;
102.                    return false;
103.                }
104.            }
105.            else
106.            {
107.                cout << "Erreur: } attendu dans declarationPrime" << endl;
108.                return false;
109.            }
110.        }
111.        else
112.        {
113.            return false;
114.        }
115.    }
116.    else
117.    {
118.        return false;
119.    }
120.}
121.
122.bool Syntaxique::listeDeclarateursStruct()
123.{
124.    if(motCourant.UL == VIRG)
125.    {
126.        motCourant = lex.uniteSuivante();
127.        if(motCourant.UL == IDENT)
128.        {
129.            motCourant = lex.uniteSuivante();
130.            return listeDeclarateursStruct();
131.        }
132.        else
133.        {
134.            cout << "Erreur: , attendu dans listeDeclarateursStruct" << endl;
135.            return false;
```

```
136.         }
137.     }
138.     else
139.     {
140.         return true;
141.     }
142. }
143. 
144. bool Syntaxique::structure()
145. {
146.     if (motCourant.UL == IDENT)
147.     {
148.         motCourant = lex.uniteSuivante();
149.         if (motCourant.UL == ACC_OUV)
150.         {
151.             motCourant = lex.uniteSuivante();
152.             if (listeAttributs()) // Mot en avance
153.             {
154.                 if (motCourant.UL == ACC_FER)
155.                 {
156.                     motCourant = lex.uniteSuivante();
157.                     if (motCourant.UL == SEP)
158.                     {
159.                         motCourant = lex.uniteSuivante();
160.                         return true;
161.                     }
162.                     else
163.                     {
164.                         cout << "Erreur: ; attendu dans structure" << endl;
165.                         return false;
166.                     }
167.                 }
168.                 else
169.                 {
170.                     cout << "Erreur: } attendu dans structure" << endl;
171.                     return false;
172.                 }
173.             }
174.             else // Erreur: liste d'attribut false
175.                 return false;
176.         }
177.         else
178.         {
179.             cout << "Erreur: ; attendu dans listeAttributs" << endl;
180.             return false;
181.         }
182.     }
183.     else // Erreur: identificateur incorrect
184.     {
185.         return false;
186.     }
187. }
188. 
189. bool Syntaxique::listeAttributs()
190. {
191.     if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
192.     {
193.         motCourant = lex.uniteSuivante();
194.         if (declarateur())
195.         {
196.             if (motCourant.UL == SEP)
197.             {
198.                 motCourant = lex.uniteSuivante();
199.                 return listeAttributsPrime();
200.             }
201.             else
202.             {
203.                 cout << "Erreur: ; attendu dans listeAttributs" << endl;
204.                 return false;
```

```
205.             }
206.         }
207.         else // gestion erreur au sein de la fct declarateur
208.         {
209.             return false;
210.         }
211.     }
212.     else
213.     {
214.         cout << "Erreur: mot clé non reconnu" << endl;
215.         return false;
216.     }
217. }
218. 
219. bool Syntaxique::listeAttributsPrime()
220. {
221.     if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
222.     {
223.         motCourant = lex.uniteSuivante();
224.         if (declarateur())
225.         {
226.             if (motCourant.UL == SEP)
227.             {
228.                 motCourant = lex.uniteSuivante();
229.                 return listeAttributsPrime();
230.             }
231.             else
232.             {
233.                 cout << "Erreur: ; attendu dans listeAttributsPrime" << endl;
234.                 return false;
235.             }
236.         }
237.         else // gestion erreur au sein de la fct declarateur
238.         {
239.             return false;
240.         }
241.     }
242.     else // epsilon
243.     {
244.         return true;
245.     }
246. }
247. 
248. bool Syntaxique::declarateur()
249. {
250.     if (motCourant.UL == IDENT)
251.     {
252.         motCourant = lex.uniteSuivante();
253.         return declarateurPrime();
254.     }
255.     else // Erreur: identificateur incorrect
256.     {
257.         return false;
258.     }
259. }
260. 
261. bool Syntaxique::declarateurPrime()
262. {
263.     if (motCourant.UL == CRO_OUV)
264.     {
265.         motCourant = lex.uniteSuivante();
266.         if (motCourant.UL == CONST_INT)
267.         {
268.             motCourant = lex.uniteSuivante();
269.             if (motCourant.UL == CRO_FER)
270.             {
271.                 motCourant = lex.uniteSuivante();
272.                 return true;
273.             }
```

```
274.             else
275.             {
276.                 cout << "Erreur: ; attendu dans declarateurPrime" << endl;
277.                 return false;
278.             }
279.         }
280.         else
281.         {
282.             cout << "Erreur: constante incorrecte" << endl;
283.             return false;
284.         }
285.     }
286.     else // epsilon
287.     {
288.         return true;
289.     }
290. }
291. 
292. bool Syntaxique::programmePrime()
293. {
294.     if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
295.     {
296.         motCourant = lex.uniteSuivante();
297.         if (motCourant.UL == IDENT)
298.         {
299.             motCourant = lex.uniteSuivante();
300.             return programmeSeconde();
301.         }
302.         else // Erreur: identifiant incorrect
303.             return false;
304.     }
305.     else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("extern"))
306.     {
307.         motCourant = lex.uniteSuivante();
308.         return programmeTierce();
309.     }
310.     else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("void"))
311.     {
312.         motCourant = lex.uniteSuivante();
313.         if (motCourant.UL == IDENT)
314.         {
315.             motCourant = lex.uniteSuivante();
316.             if (motCourant.UL == PAR_OUV)
317.             {
318.                 motCourant = lex.uniteSuivante();
319.                 if (listeParams())
320.                 {
321.                     if (motCourant.UL == PAR_FER)
322.                     {
323.                         motCourant = lex.uniteSuivante();
324.                         if (motCourant.UL == ACC_OUV)
325.                         {
326.                             motCourant = lex.uniteSuivante();
327.                             if (listeDeclarations())
328.                             {
329.                                 if (listeInstructions())
330.                                 {
331.                                     if (motCourant.UL == ACC_FER)
332.                                     {
333.                                         motCourant = lex.uniteSuivante();
334.                                         return listeFonctionsPrime();
335.                                     }
336.                                     else
337.                                     {
338.                                         cout << "Erreur: } attendu dans programmePrime" << endl
    ;
339.                                         return false;
340.                                     }
341.                                 }
```

```
342.                                 else // Erreur: liste instructions
343.                                 {
344.                                     return false;
345.                                 }
346.                             }
347.                             else // Erreur: liste Parametres
348.                             {
349.                                 return false;
350.                             }
351.                         }
352.                         else // Erreur: { attendu
353.                         {
354.                             cout << "Erreur: { attendu dans programmePrime" << endl;
355.                             return false;
356.                         }
357.                     }
358.                     else // Erreur: ) attendu
359.                     {
360.                         cout << "Erreur: ) attendu dans programmePrime" << endl;
361.                         return false;
362.                     }
363.                 }
364.                 else // erreur listeParamas
365.                 {
366.                     return false;
367.                 }
368.             }
369.             else
370.             {
371.                 cout << "Erreur: ( attendu dans programmePrime" << endl;
372.                 return false;
373.             }
374.         }
375.         else // Erreur: identificateur incorrect
376.         {
377.             cout << "Erreur: identificateur attendu dans programmePrime" << endl;
378.             return false;
379.         }
380.     }
381.
382.     else if(motCourant.UL ==  MOT_CLE && motCourant.attribut == lex.hashMotCle("struct"))
383.     {
384.         if(blocStruct())
385.         {
386.             return programmePrime();
387.         }
388.         else
389.         {
390.             return false;
391.         }
392.     }
393.     else // erreur programmePrime
394.     {
395.         return false;
396.     }
397. }
398.
399. bool Syntaxique::programmeSeconde()
400. {
401.     if (motCourant.UL == PAR_OUV)
402.     {
403.         motCourant = lex.uniteSuivante();
404.         if (listeParams())
405.         {
406.             if (motCourant.UL == PAR_FER)
407.             {
408.                 motCourant = lex.uniteSuivante();
409.                 if (motCourant.UL == ACC_OUV)
410.                 {
```

```
411.                     motCourant = lex.uniteSuivante();
412.                     if (listeDeclarations())
413.                     {
414.                         if (listeInstructions())
415.                         {
416.                             if (motCourant.UL == ACC_FER)
417.                             {
418.                                 motCourant = lex.uniteSuivante();
419.                                 return listeFonctionsPrime();
420.                             }
421.                             else
422.                             {
423.                                 cout << "Erreur: } attendu dans programmeSeconde" << endl;
424.                                 return false;
425.                             }
426.                         }
427.                         else
428.                         {
429.                             return false;
430.                         }
431.                     }
432.                     else
433.                     {
434.                         return false;
435.                     }
436.                 }
437.                 else
438.                 {
439.                     cout << "Erreur: { attendu dans programmeSeconde" << endl;
440.                     return false;
441.                 }
442.             }
443.             else
444.             {
445.                 cout << "Erreur: ) attendu dans programmeSeconde" << endl;
446.             }
447.         }
448.         else
449.         {
450.             return false;
451.         }
452.     }
453.     else if (declarateurPrime())
454.     {
455.         if (listeDeclarateursPrime())
456.         {
457.             if (motCourant.UL == SEP)
458.             {
459.                 motCourant = lex.uniteSuivante();
460.                 return programmePrime();
461.             }
462.             else
463.             {
464.                 cout << "Erreur: ; attendu dans programmeSeconde" << endl;
465.                 return false;
466.             }
467.         }
468.         else
469.         {
470.             return false;
471.         }
472.     }
473.     else
474.     {
475.         return false;
476.     }
477. }
478.
479.
```

```
480.bool Syntaxique::programmeTierce()
481.{
482.    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("void"))
483.    {
484.        motCourant = lex.uniteSuivante();
485.        if(motCourant.UL == IDENT)
486.        {
487.            motCourant = lex.uniteSuivante();
488.            if(motCourant.UL == PAR_OUV)
489.            {
490.                motCourant = lex.uniteSuivante();
491.                if(listeParams())
492.                {
493.                    if(motCourant.UL == PAR_FER)
494.                    {
495.                        motCourant = lex.uniteSuivante();
496.                        if(motCourant.UL == SEP)
497.                        {
498.                            motCourant = lex.uniteSuivante();
499.                            return listeFonctionsPrime();
500.                        }
501.                        else
502.                        {
503.                            cout << "Erreur: ; attendu dans programmeTierce" << endl;
504.                            return false;
505.                        }
506.                    }
507.                    else
508.                    {
509.                        cout << "Erreur: ) attendu dans programmeTierce" << endl;
510.                        return false;
511.                    }
512.                }
513.                else
514.                {
515.                    return false;
516.                }
517.            }
518.            else
519.            {
520.                cout << "Erreur: ( attendu dans programmeTierce" << endl;
521.                return false;
522.            }
523.        }
524.        else
525.        {
526.            cout << "Erreur: indentificateur attendu dans programmeTierce" << endl;
527.            return false;
528.        }
529.    }
530.
531.    else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
532.    {
533.        motCourant = lex.uniteSuivante();
534.        if(motCourant.UL == IDENT)
535.        {
536.            motCourant = lex.uniteSuivante();
537.            if(motCourant.UL == PAR_OUV)
538.            {
539.                motCourant = lex.uniteSuivante();
540.                if(listeParams())
541.                {
542.                    if(motCourant.UL == PAR_FER)
543.                    {
544.                        motCourant = lex.uniteSuivante();
545.                        if(motCourant.UL == SEP)
546.                        {
547.                            motCourant = lex.uniteSuivante();
548.                            return listeFonctionsPrime();
```

```
549.                         }
550.                         else
551.                         {
552.                             cout << "Erreur: ; attendu dans programmeTierce" << endl;
553.                             return false;
554.                         }
555.                     }
556.                     else
557.                     {
558.                         cout << "Erreur: ) attendu dans programmeTierce" << endl;
559.                         return false;
560.                     }
561.                 }
562.                 else
563.                 {
564.                     return false;
565.                 }
566.             }
567.             else
568.             {
569.                 cout << "Erreur: ( attendu dans programmeTierce" << endl;
570.                 return false;
571.             }
572.         }
573.         else
574.         {
575.             cout << "Erreur: indentificateur attendu dans programmeTierce" << endl;
576.             return false;
577.         }
578.     }
579.     else
580.     {
581.         return false;
582.     }
583.}
584.
585.bool Syntaxique::listeParams()
586.{
587.     if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
588.     {
589.
590.         if (parametre())
591.         {
592.             return listeParamsPrime();
593.         }
594.         else
595.             return false;
596.     }
597.     else
598.     {
599.         return true;
600.     }
601.}
602.
603.bool Syntaxique::listeParamsPrime()
604.{
605.     if (motCourant.UL == VIRG)
606.     {
607.         motCourant = lex.uniteSuivante();
608.         if (parametre())
609.         {
610.             return listeParamsPrime();
611.         }
612.         else
613.             return false;
614.     }
615.     else
616.         return true;
617.}
```

```
618.
619.bool Syntaxique::parametre()
620.{
621.    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
622.    {
623.        motCourant = lex.uniteSuivante();
624.        if (motCourant.UL == IDENT)
625.        {
626.            motCourant = lex.uniteSuivante();
627.            return true;
628.        }
629.        else
630.        {
631.            cout << "Erreur: identificateur attendu parametre" << endl;
632.            return false;
633.        }
634.    }
635.    else
636.    {
637.        cout << "Erreur: int attendu dans parametre" << endl;
638.        return false;
639.    }
640.}
641.
642.bool Syntaxique::declaration()
643.{
644.    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))
645.    {
646.        motCourant = lex.uniteSuivante();
647.        if (listeDeclarateurs())
648.        {
649.            if (motCourant.UL == SEP)
650.            {
651.                motCourant = lex.uniteSuivante();
652.                return true;
653.            }
654.            else
655.            {
656.                cout << "Erreur: ; attendu dans declaration" << endl;
657.                return false;
658.            }
659.        }
660.        else
661.        {
662.            return false;
663.        }
664.    }
665.    else if(motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("struct") )
666.    {
667.        motCourant = lex.uniteSuivante();
668.        if(motCourant.UL == IDENT)
669.        {
670.            motCourant = lex.uniteSuivante();
671.            return declarationPrime();
672.        }
673.        else{
674.            cout << "Erreur: identificateur attendu dans declaration" << endl;
675.            return false;
676.        }
677.    }
678.    else{
679.        return false;
680.    }
681.}
682.
683.bool Syntaxique::listeDeclarations()
684.{
685.    if (motCourant.UL == MOT_CLE && (motCourant.attribut == lex.hashMotCle("int") ||
686.                                     motCourant.attribut == lex.hashMotCle("struct")))
```

```
    {
        if (declaration())
        {
            return listeDeclarations();
        }
        else
            return false;
    }
    else
        return true;
}

bool Syntaxique::listeInstructions()
{
    if ((motCourant.UL == MOT_CLE && (motCourant.attribut == lex.hashMotCle("for") ||
                                      motCourant.attribut == lex.hashMotCle("while") ||
                                      motCourant.attribut == lex.hashMotCle("if") ||
                                      motCourant.attribut == lex.hashMotCle("return"))) ||
            motCourant.UL == IDENT ||
            motCourant.UL == ACC_OUV)
    {
        if (instruction())
        {
            return listeInstructions();
        }
        else
            return false;
    }
    return true;
}

////////////////////////////////////////////////////////tested////////////////////////////////////////////////

bool Syntaxique::instruction()
{
    if (motCourant.UL == MOT_CLE && (motCourant.attribut == lex.hashMotCle("for") || motCourant.attribut == lex.hashMotCle("while")))
    {
        //pas d'avancement : deux types d'iterrations
        return iteration();
    }
    else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("if"))
    {
        motCourant = lex.uniteSuivante();
        return selection();
    }
    else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("return"))
        ////////////////////////////////////////////////////////////////////tested////////////////////////////////////////////////////
    {
        motCourant = lex.uniteSuivante();
        return saut();
    }
    else if (motCourant.UL == ACC_OUV)
    {
        ////////////////////////////////////////////////////////////////////tested////////////////////////////////////////////////////
        motCourant = lex.uniteSuivante();
        return bloc();
    }
    else if (motCourant.UL == IDENT)
    {
        ////////////////////////////////////////////////////////////////////tested////////////////////////////////////////////////////
        motCourant = lex.uniteSuivante();
        return instructionPrime();
    }
    else
```

```
751.        return false;
752.}
753.
754.bool Syntaxique::iteration()
755.{
756.    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("for"))
757.    {
758.        motCourant = lex.uniteSuivante();
759.        if (motCourant.UL == PAR_OUV)
760.        {
761.            motCourant = lex.uniteSuivante();
762.            if (affectation())
763.            {
764.                if (motCourant.UL == SEP)
765.                {
766.                    motCourant = lex.uniteSuivante();
767.                    if (condition())
768.                    {
769.                        if (motCourant.UL == SEP)
770.                        {
771.                            motCourant = lex.uniteSuivante();
772.                            if (affectation())
773.                            {
774.                                if (motCourant.UL == PAR_FER)
775.                                {
776.                                    motCourant = lex.uniteSuivante();
777.                                    if (instruction())
778.                                    {
779.                                        return true;
780.                                    }
781.                                    else
782.                                    {
783.                                        return false;
784.                                    }
785.                                }
786.                                else
787.                                {
788.                                    cout << "Erreur: ) attendu dans iteration" << endl;
789.                                    return false;
790.                                }
791.                            }
792.                            else
793.                            {
794.                                return false;
795.                            }
796.                        }
797.                        else
798.                        {
799.                            cout << "Erreur: ; attendu dans iteration" << endl;
800.                            return false;
801.                        }
802.                    }
803.                    else
804.                    {
805.                        return false;
806.                    }
807.                }
808.                else
809.                {
810.                    cout << "Erreur: ; attendu dans iteration" << endl;
811.                    return false;
812.                }
813.            }
814.            else
815.            {
816.                return false;
817.            }
818.        }
819.        else
```

```
820.         {
821.             cout << "Erreur: ( attendu dans iteration" << endl;
822.             return false;
823.         }
824.     }
825.     else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("while"))
826.     {
827.         motCourant = lex.uniteSuivante();
828.         if (motCourant.UL == PAR_OUV)
829.         {
830.             motCourant = lex.uniteSuivante();
831.             if (condition())
832.             {
833.                 if (motCourant.UL == PAR_FER)
834.                 {
835.                     motCourant = lex.uniteSuivante();
836.                     return instruction();
837.                 }
838.                 else
839.                 {
840.                     cout << "Erreur: ) attendu dans iteration" << endl;
841.                     return false;
842.                 }
843.             }
844.             else
845.                 return false;
846.         }
847.         else
848.         {
849.             cout << "Erreur: ( attendu dans iteration" << endl;
850.             return false;
851.         }
852.     }
853.     else
854.         return false;
855. }
856. 
857. bool Syntaxique::saut()
858. {
859.     return sautPrime();
860. }
861. 
862. bool Syntaxique::sautPrime()
863. {
864.     if (expression())
865.     {
866.         if (motCourant.UL == SEP)
867.         {
868.             motCourant = lex.uniteSuivante();
869.             return true;
870.         }
871.         else
872.         {
873.             cout << "Erreur: ; attendu dans sautprime" << endl;
874.             return false;
875.         }
876.     }
877.     else if (motCourant.UL == SEP)
878.     {
879.         motCourant = lex.uniteSuivante();
880.         return true;
881.     }
882.     else
883.         return false;
884. }
885. 
886. bool Syntaxique::selection()
887. {
888.     if (motCourant.UL == PAR_OUV)
```

```
889.    {
890.        motCourant = lex.uniteSuivante();
891.        if (condition())
892.        {
893.            if (motCourant.UL == PAR_FER)
894.            {
895.                motCourant = lex.uniteSuivante();
896.                if (instruction())
897.                {
898.                    return selectionPrime();
899.                }
900.                else
901.                {
902.                    return false;
903.                }
904.            }
905.            else
906.            {
907.                cout << "Erreur: ) attendu dans selection" << endl;
908.                return false;
909.            }
910.        }
911.        else
912.        {
913.            return false;
914.        }
915.    }
916.    else
917.    {
918.        cout << "Erreur: ( attendu dans selection" << endl;
919.        return false;
920.    }
921.}
922.
923.bool Syntaxique::selectionPrime()
924.{
925.    if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("else"))
926.    {
927.        motCourant = lex.uniteSuivante();
928.        return instruction();
929.    }
930.    else //epsilon
931.    {
932.        return true;
933.    }
934.}
935.
936.bool Syntaxique::listeDeclarateurs()
937.{
938.    if (declarateur())
939.    {
940.        return listeDeclarateursPrime();
941.    }
942.    else
943.        return false;
944.}
945.
946.bool Syntaxique::listeDeclarateursPrime()
947.{
948.    if (motCourant.UL == VIRG)
949.    {
950.        motCourant = lex.uniteSuivante();
951.        if (declarateur())
952.        {
953.            return listeDeclarateursPrime();
954.        }
955.
956.    }
957.    return true;
```

```
958.}
959.
960.bool Syntaxique::bloc()
961.{
962.    if (listeInstructions())
963.    {
964.        if (motCourant.UL == ACC_FER)
965.        {
966.            motCourant = lex.uniteSuivante();
967.            return true;
968.        }
969.        else
970.        {
971.            cout << "Erreur: } attendu dans bloc" << endl;
972.            return false;
973.        }
974.    }
975.    else
976.    {
977.        return false;
978.    }
979.}
980.
981.bool Syntaxique::instructionPrime()
982.{
983.    if (motCourant.UL == CRO_OUV)
984.    {
985.        motCourant = lex.uniteSuivante();
986.        if (expression())
987.        {
988.            if (motCourant.UL == CRO_FER)
989.            {
990.                motCourant = lex.uniteSuivante();
991.                if (motCourant.UL == AFFEC)
992.                {
993.                    motCourant = lex.uniteSuivante();
994.                    if (expression())
995.                    {
996.                        if (motCourant.UL == SEP)
997.                        {
998.                            motCourant = lex.uniteSuivante();
999.                            return true;
1000.                           }
1001.                           else
1002.                           {
1003.                               cout << "Erreur: ; attendu dans instruction prime" << endl;
1004.                               return false;
1005.                           }
1006.                       }
1007.                       else
1008.                       {
1009.                           return false;
1010.                       }
1011.                   }
1012.                   else
1013.                   {
1014.                       cout << "Erreur: = attendu dans instruction prime" << endl;
1015.                       return false;
1016.                   }
1017.               }
1018.               else
1019.               {
1020.                   cout << "Erreur: ] attendu dans instruction prime " << endl;
1021.                   return false;
1022.               }
1023.           }
1024.           else
1025.               return false;
```

```
1026.           }
1027.           else if (motCourant.UL == PAR_OUV)
1028.           {
1029.               motCourant = lex.uniteSuivante();
1030.               if (listeExpressions())
1031.               {
1032.                   if (motCourant.UL == PAR_FER)
1033.                   {
1034.                       motCourant = lex.uniteSuivante();
1035.                       if (motCourant.UL == SEP)
1036.                       {
1037.                           motCourant = lex.uniteSuivante();
1038.                           return true;
1039.                       }
1040.                       else
1041.                       {
1042.                           cout << "Erreur: ; attendu dans instruction prime" << endl;
1043.                           return false;
1044.                       }
1045.                   }
1046.                   else
1047.                   {
1048.                       cout << "Erreur: ) attendu dans instruction prime" << endl;
1049.                       return false;
1050.                   }
1051.               }
1052.               else
1053.               {
1054.                   return false;
1055.               }
1056.           }
1057.           else if (motCourant.UL == AFFEC)
1058.           {
1059.               motCourant = lex.uniteSuivante();
1060.               if (expression())
1061.               {
1062.                   if (motCourant.UL == SEP)
1063.                   {
1064.                       motCourant = lex.uniteSuivante();
1065.                       return true;
1066.                   }
1067.                   else
1068.                   {
1069.                       cout << "Erreur: ; attendu dans instruction prime" << endl;
1070.                       return false;
1071.                   }
1072.               }
1073.               else
1074.                   return false;
1075.           }
1076.           else
1077.           {
1078.               return false;
1079.           }
1080.       }
1081.       bool Syntaxique::affectation()
1082.       {
1083.           if (variable())
1084.           {
1085.               if (motCourant.UL == AFFEC)
1086.               {
1087.                   motCourant = lex.uniteSuivante();
1088.                   return expression();
1089.               }
1090.               else
1091.               {
1092.                   cout << "Erreur: = attendu dans affectation" << endl;
1093.                   return false;
1094.               }
```

```
1095.           }
1096.           else
1097.           {
1098.               return false;
1099.           }
1100.       }
1101.
1102.       bool Syntaxique::variable()
1103.       {
1104.           if (motCourant.UL == IDENT)
1105.           {
1106.               motCourant = lex.uniteSuivante();
1107.               return variablePrime();
1108.           }
1109.           else
1110.               return false;
1111.       }
1112.
1113.       bool Syntaxique::variablePrime()
1114.       {
1115.           if (motCourant.UL == CRO_OUV)
1116.           {
1117.               motCourant = lex.uniteSuivante();
1118.               if (expression())
1119.               {
1120.                   if (motCourant.UL == CRO_FER)
1121.                   {
1122.                       motCourant = lex.uniteSuivante();
1123.                       return true;
1124.                   }
1125.                   else
1126.                   {
1127.                       cout << "Erreur: ] attendu dans variablePrime" << endl;
1128.                       return false;
1129.                   }
1130.               }
1131.               else
1132.                   return false;
1133.           }
1134.           else if (motCourant.UL == POINT)
1135.           {
1136.               motCourant = lex.uniteSuivante();
1137.               if (motCourant.UL == IDENT)
1138.               {
1139.                   motCourant = lex.uniteSuivante();
1140.                   return true;
1141.               }
1142.               else
1143.               {
1144.                   cout << "Erreur: Identificateur attendu dans variablePrime" << endl;
1145.                   return false;
1146.               }
1147.           }
1148.           else
1149.               return true;
1150.       }
1151.
1152.       bool Syntaxique::condition()
1153.       {
1154.           if (motCourant.UL == NON)
1155.           {
1156.               motCourant = lex.uniteSuivante();
1157.               if (motCourant.UL == PAR_OUV)
1158.               {
1159.                   motCourant = lex.uniteSuivante();
1160.                   if (condition())
1161.                   {
1162.                       if (motCourant.UL == PAR_FER)
1163.                       {
```

```
1164.                           motCourant = lex.uniteSuivante();
1165.                           return conditionPrime();
1166.                       }
1167.                       else
1168.                       {
1169.                           cout << "Erreur: ) attendue dans condition" << endl;
1170.                           return false;
1171.                       }
1172.                   }
1173.                   return false;
1174.               }
1175.               else
1176.               {
1177.                   cout << "Erreur: ( attendue dans condition" << endl;
1178.                   return false;
1179.               }
1180.           }
1181.           else if (motCourant.UL == PAR_OUV)
1182.           {
1183.               motCourant = lex.uniteSuivante();
1184.               if (condition())
1185.               {
1186.                   if (motCourant.UL == PAR_FER)
1187.                   {
1188.                       motCourant = lex.uniteSuivante();
1189.                       return conditionPrime();
1190.                   }
1191.                   cout << "Erreur: ) attendue dans condition" << endl;
1192.                   return false;
1193.               }
1194.
1195.               else
1196.               {
1197.                   return false;
1198.               }
1199.           }
1200.
1201.           else if (expression())
1202.           {
1203.               if (binaryComp())
1204.               {
1205.                   if (expression())
1206.                   {
1207.                       return conditionPrime();
1208.                   }
1209.                   else
1210.                   {
1211.                       return false;
1212.                   }
1213.               }
1214.               else
1215.               {
1216.                   cout << "Erreur: binaryComp attendue dans condition" << endl;
1217.                   return false;
1218.               }
1219.           }
1220.
1221.           else
1222.           {
1223.               return false;
1224.           }
1225.       }
1226.
1227.       bool Syntaxique::conditionPrime()
1228.       {
1229.           if (binaryRel())
1230.           {
1231.               if (condition())
1232.               {
```

```
1233.                   return conditionPrime();
1234.               }
1235.               else
1236.               {
1237.                   return false;
1238.               }
1239.           }
1240.           else
1241.           {
1242.               return true;
1243.           }
1244.       }
1245.
1246.       bool Syntaxique::binaryOp()
1247.       {
1248.           if (motCourant.UL == PLUS)
1249.           {
1250.               motCourant = lex.uniteSuivante();
1251.               return true;
1252.           }
1253.           else if (motCourant.UL == MOINS)
1254.           {
1255.               motCourant = lex.uniteSuivante();
1256.               return true;
1257.           }
1258.           else if (motCourant.UL == MULT)
1259.           {
1260.               motCourant = lex.uniteSuivante();
1261.               return true;
1262.           }
1263.           else if (motCourant.UL == DIV)
1264.           {
1265.               motCourant = lex.uniteSuivante();
1266.               return true;
1267.           }
1268.           else if (motCourant.UL == DEC_DROITE)
1269.           {
1270.               motCourant = lex.uniteSuivante();
1271.               return true;
1272.           }
1273.           else if (motCourant.UL == DEC_GAUCHE)
1274.           {
1275.               motCourant = lex.uniteSuivante();
1276.               return true;
1277.           }
1278.           else if (motCourant.UL == BIN_ET)
1279.           {
1280.               motCourant = lex.uniteSuivante();
1281.               return true;
1282.           }
1283.           else if (motCourant.UL == BIN_OU)
1284.           {
1285.               motCourant = lex.uniteSuivante();
1286.               return true;
1287.           }
1288.           else
1289.           {
1290.               return false;
1291.           }
1292.       }
1293.
1294.       bool Syntaxique::binaryRel()
1295.       {
1296.           if (motCourant.UL == ET)
1297.           {
1298.               motCourant = lex.uniteSuivante();
1299.               return true;
1300.           }
1301.           else if (motCourant.UL == OU)
```

```
1302.           {
1303.               motCourant = lex.uniteSuivante();
1304.               return true;
1305.           }
1306.           else
1307.           {
1308.               return false;
1309.           }
1310.       }
1311.
1312.       bool Syntaxique::binaryComp()
1313.       {
1314.           if (motCourant.UL == INF)
1315.           {
1316.               motCourant = lex.uniteSuivante();
1317.               return true;
1318.           }
1319.           else if (motCourant.UL == SUP)
1320.           {
1321.               motCourant = lex.uniteSuivante();
1322.               return true;
1323.           }
1324.           else if (motCourant.UL == SUPEGAL)
1325.           {
1326.               motCourant = lex.uniteSuivante();
1327.               return true;
1328.           }
1329.           else if (motCourant.UL == INFEGAL)
1330.           {
1331.               motCourant = lex.uniteSuivante();
1332.               return true;
1333.           }
1334.           else if (motCourant.UL == EGAL)
1335.           {
1336.               motCourant = lex.uniteSuivante();
1337.               return true;
1338.           }
1339.           else if (motCourant.UL == DIFF)
1340.           {
1341.               motCourant = lex.uniteSuivante();
1342.               return true;
1343.           }
1344.           else
1345.           {
1346.               return false;
1347.           }
1348.       }
1349.
1350.       bool Syntaxique::listeExpressions()
1351.       {
1352.           if (motCourant.UL == PAR_OUV || motCourant.UL == MOINS || motCourant.UL == IDENT ||
      motCourant.UL == CONST_INT)
1353.           {
1354.               if (expression())
1355.               {
1356.                   return listeExpressionsPrime();
1357.               }
1358.               else
1359.               {
1360.                   return false;
1361.               }
1362.           }
1363.           else
1364.           {
1365.               return true;
1366.           }
1367.       }
1368.
1369.       bool Syntaxique::listeExpressionsPrime()
```

```
1370.       {
1371.           if(motCourant.UL == VIRG)
1372.           {
1373.               motCourant = lex.uniteSuivante();
1374.               if(expression())
1375.               {
1376.                   return listeExpressionsPrime();
1377.               }
1378.               else
1379.               {
1380.                   return false;
1381.               }
1382.           }
1383.           else
1384.               return true;
1385.       }
1386.
1387.       bool Syntaxique::listeFonctions()
1388.       {
1389.           if (fonction())
1390.           {
1391.               motCourant = lex.uniteSuivante();
1392.               return listeFonctionsPrime();
1393.           }
1394.           else
1395.           {
1396.               return false;
1397.           }
1398.       }
1399.
1400.       bool Syntaxique::listeFonctionsPrime()
1401.       {
1402.           if (motCourant.UL == MOT_CLE && (motCourant.attribut == lex.hashMotCle("int")||motC
    ourant.attribut == lex.hashMotCle("void")||motCourant.attribut == lex.hashMotCle("extern")))
1403.           {
1404.               if(fonction())
1405.               {
1406.                   return listeFonctionsPrime();
1407.               }
1408.               else
1409.               {
1410.                   cout << "Erreur: fonction mal definit" << endl;
1411.                   return false ;
1412.               }
1413.           }
1414.           else if (motCourant.UL == ENDOF)
1415.           {
1416.               return true;
1417.           }
1418.           else
1419.           {
1420.               return false;
1421.           }
1422.       }
1423.
1424.       bool Syntaxique::fonction()
1425.       {
1426.           if (type())
1427.           {
1428.               if (motCourant.UL == IDENT)
1429.               {
1430.                   motCourant = lex.uniteSuivante();
1431.                   if (motCourant.UL == PAR_OUV)
1432.                   {
1433.                       motCourant = lex.uniteSuivante();
1434.                       if (listeParams())
1435.                       {
1436.                           if (motCourant.UL == PAR_FER)
1437.                           {
```

```
1438.                                 motCourant = lex.uniteSuivante();
1439.                                 if (motCourant.UL == ACC_OUV)
1440.                                 {
1441.                                     motCourant = lex.uniteSuivante();
1442.                                     if (listeDeclarations())
1443.                                     {
1444.                                         if(listeInstructions())
1445.                                         {
1446.                                             if(motCourant.UL == ACC_FER)
1447.                                             {
1448.                                                 motCourant = lex.uniteSuivante();
1449.                                                 return true;
1450.                                             }
1451.                                             else
1452.                                             {
1453.                                                 cout <<"Erreur: } attendue dans fonction" << en
    dl;
1454.                                                 return false;
1455.                                             }
1456.                                         }
1457.                                         else
1458.                                         {
1459.                                             return false;
1460.                                         }
1461.                                     }
1462.                                     else
1463.                                     {
1464.                                         return false;
1465.                                     }
1466.                                 }
1467.                                 else
1468.                                 {
1469.                                     cout << "Erreur: { attendue dans fonction" << endl;
1470.                                     return false;
1471.                                 }
1472.                             }
1473.                             else
1474.                             {
1475.                                 cout << "Erreur: ) attendue dans fonction" << endl;
1476.                                 return false;
1477.                             }
1478.                         }
1479.                         else
1480.                         {
1481.                             return false;
1482.                         }
1483.                     }
1484.                     else
1485.                     {
1486.                         cout << "Erreur: ( attendue dans fonction" << endl;
1487.                         return false;
1488.                     }
1489.                 }
1490.                 else
1491.                 {
1492.                     cout << "Erreur: identificateur attendu dans fonction" << endl;
1493.                     return false;
1494.                 }
1495.             }
1496.
1497.             else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("extern"
    ))
1498.             {
1499.                 motCourant = lex.uniteSuivante();
1500.                 if (type())
1501.                 {
1502.                     if (motCourant.UL == IDENT)
1503.                     {
1504.                         motCourant = lex.uniteSuivante();
```

```
1505.                       if (motCourant.UL == PAR_OUV)
1506.                       {
1507.                           motCourant = lex.uniteSuivante();
1508.                           if (listeParams())
1509.                           {
1510.                               if (motCourant.UL == PAR_FER)
1511.                               {
1512.                                   motCourant = lex.uniteSuivante();
1513.                                   if (motCourant.UL == SEP)
1514.                                   {
1515.                                       motCourant = lex.uniteSuivante();
1516.                                       return true;
1517.                                   }
1518.                                   else
1519.                                   {
1520.                                       cout << "Erreur: ; attendu dans fonction" << endl;
1521.                                       return false;
1522.                                   }
1523.                               }
1524.                               else
1525.                               {
1526.                                   cout << "Erreur: ) attendu dans fonction" << endl;
1527.                                   return false;
1528.                               }
1529.                           }
1530.                           else
1531.                           {
1532.                               return false;
1533.                           }
1534.                       }
1535.                       else
1536.                       {
1537.                           cout << "Erreur: ( attendu dans fonction" << endl;
1538.                           return false;
1539.                       }
1540.                   }
1541.                   else
1542.                   {
1543.                       cout << "Erreur: identificateur attendu dans fonction" << endl;
1544.                       return false;
1545.                   }
1546.               }
1547.           }
1548.
1549.           else
1550.           {
1551.               return false;
1552.           }
1553.       }
1554.
1555.       bool Syntaxique::type()
1556.       {
1557.           if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("void"))
1558.           {
1559.               motCourant = lex.uniteSuivante();
1560.               return true;
1561.           }
1562.           else if (motCourant.UL == MOT_CLE && motCourant.attribut == lex.hashMotCle("int"))

1563.           {
1564.               motCourant = lex.uniteSuivante();
1565.               return true;
1566.           }
1567.
1568.           else
1569.           {
1570.               return false;
1571.           }
1572.       }
```

```
1573.
1574.       bool Syntaxique::expression()
1575.       {
1576.           if(motCourant.UL == PAR_OUV)
1577.           {
1578.               motCourant = lex.uniteSuivante();
1579.               if(expression())
1580.               {
1581.                   if(motCourant.UL == PAR_FER)
1582.                   {
1583.                       motCourant = lex.uniteSuivante();
1584.                       return true;
1585.                   }
1586.                   else
1587.                   {
1588.                       cout << "Erreur: ) attendu dans expression" << endl;
1589.                       return false;
1590.                   }
1591.
1592.               }
1593.               else
1594.               {
1595.                   return false;
1596.               }
1597.           }
1598.           else if(motCourant.UL == IDENT)
1599.           {
1600.               motCourant = lex.uniteSuivante();
1601.               if(expressionPrime())
1602.               {
1603.                   return expressionSeconde();
1604.               }
1605.               else
1606.               {
1607.                   return false;
1608.               }
1609.           }
1610.           else if(motCourant.UL == CONST_INT)
1611.           {
1612.               motCourant = lex.uniteSuivante();
1613.               return expressionSeconde();
1614.           }
1615.
1616.           else if(motCourant.UL == MOINS)
1617.           {
1618.               motCourant = lex.uniteSuivante();
1619.               return expression();
1620.           }
1621.
1622.           else
1623.           {
1624.               return false;
1625.           }
1626.       }
1627.
1628.
1629.       bool Syntaxique::expressionPrime()
1630.       {
1631.           if(motCourant.UL == CRO_OUV)
1632.           {
1633.               motCourant = lex.uniteSuivante();
1634.               if(expression())
1635.               {
1636.                   if(motCourant.UL == CRO_FER)
1637.                   {
1638.                       motCourant = lex.uniteSuivante();
1639.                       return true;
1640.                   }
1641.                   else
```

```
1642.                   {
1643.                       cout << "Erreur: ] attendu dans expression" << endl;
1644.                       return false;
1645.                   }
1646.               }
1647.               else
1648.               {
1649.                   return false;
1650.               }
1651.           }
1652.           else if (motCourant.UL == POINT)
1653.           {
1654.               motCourant = lex.uniteSuivante();
1655.               if(motCourant.UL == IDENT)
1656.               {
1657.                   motCourant = lex.uniteSuivante();
1658.                   return true;
1659.               }
1660.               else
1661.               {
1662.                   cout << "Erreur: identificateur attendu dans expression" << endl;
1663.                   return false;
1664.               }
1665.           }
1666.           else if(motCourant.UL == PAR_OUV)
1667.           {
1668.               motCourant = lex.uniteSuivante();
1669.               if(listeExpressions())
1670.               {
1671.                   if(motCourant.UL == PAR_FER)
1672.                   {
1673.                       motCourant = lex.uniteSuivante();
1674.                       return true;
1675.                   }
1676.                   else
1677.                   {
1678.                       cout << "Erreur: ) attendu dans expression" << endl;
1679.                       return false;
1680.                   }
1681.               }
1682.               else
1683.               {
1684.                   return false;
1685.               }
1686.           }
1687.           else
1688.           {
1689.               return true;
1690.           }
1691.       }
1692.
1693.       bool Syntaxique::expressionSeconde()
1694.       {
1695.           if(binaryOp())
1696.           {
1697.               return expression();
1698.           }
1699.           return true;
1700.       }
```