

Projet Compilateur

Rapport



Groupe 4:

BOURRICH Yassine
HANKACH Mouatamid
SAMI Ayoub

Encadré par : M. ELGHAZI Souhail

Année universitaire 2023/2024

Table des matières

| | |
|---|----|
| Introduction générale | 3 |
| Chapitre I : Environnement technique | 4 |
| 1. Langage compilateur..... | 5 |
| 2. Langage source | 5 |
| 3. Grammaire améliorée | 5 |
| Chapitre II : Analyse Lexical | 8 |
| 1. L'alphabet | 9 |
| 1. Les unités lexicales..... | 9 |
| 2. Structure de données..... | 10 |
| 1. Diagramme d'état..... | 10 |
| 5. Codage..... | 12 |
| 6. Test..... | 14 |
| Chapitre III : Analyse Syntaxique | 17 |
| I. Grammaire LL (1) :..... | 18 |
| 2. Grammaire améliorée LL (1) | 18 |
| 2. Calcule des premiers et suivants | 20 |
| 3. Justification du LL (1)..... | 21 |
| II. Codage..... | 30 |
| 1. Le code sources du syntaxique | 30 |
| 2. Teste | 31 |
| Chapitre IV : Analyse Sémantique | 32 |
| 1. Définition..... | 33 |
| 2. Périmètre de l'analyse sémantique | 33 |
| 3. Les choix techniques..... | 33 |
| 4. Les actions sémantiques..... | 35 |
| 5. Les tests | 37 |
| Conclusion | 43 |

Liste des figures

| | |
|--|-----------|
| Figure 1 : Unités lexicales | 10 |
| Figure 2 : Type de retour uniteSuivante() | 10 |
| Figure 3 : Les mots clés | 10 |
| Figure 4 : Fonction de hashage | 10 |
| Figure 5 : Remplir le tableau des mots clés | 10 |
| Figure 6: Diagramme d'état | 12 |
| Figure 7 : Ouverture du fichier | 13 |
| Figure 8 : La fonction main et gestion de la fin du fichier | 13 |
| Figure 9 : Les fonctions utilisées en uniteSuivante() | 14 |
| Figure 10 : Code de test | 14 |
| Figure 11 : Résultat du test Figure 12 : Code erroné | 15 |
| Figure 13 : Résultat de code erroné | 16 |
| Figure 14 : Fonction main | 31 |

Introduction générale

Dans le cadre de module « Théorie de compilation », nous sommes concernés de réaliser un compilateur afin de concrétiser ce que nous avons appris.

Le but de ce projet est de réaliser un petit compilateur d'un sous-ensemble du langage C appelé 'C--' vers un langage intermédiaire lui aussi un sous-ensemble de C appelé 'intC' avec quelques optimisations.

Comme nous savons un compilateur se définit par :

- Une spécification de son langage source et de son langage objet.
- Une spécification du processus de traduction de l'un en l'autre.

Alors, pour ce premier livrable nous faisons l'analyse lexicale.

Chapitre I :

Environnement technique

1. Langage compilateur

Le compilateur que nous réalisons est un compilateur C. En effet, le langage avec lequel nous le programmions est C++.

Nous choisissons ce langage car :

- **Il est adapté pour développer un programme informatique et proche de langage machine.**
- Ses programmes sont **rapides** et **interagissent avec le matériel.**
- **Il a été conçu à l'origine comme un langage proche du processeur qui peut être facilement compilé** tout en conservant de **bonnes performances.**

2. Langage source

Ce compilateur traduit un programme écrit dans le langage source C- - en un langage intermédiaire intC.

En effet, La syntaxe du langage C-- emprunte beaucoup à C, d'où le nom qui suggère que c'est essentiellement un sous-ensemble de C, de la même manière que C++ est un sur-ensemble du C.

Le langage est conçu comme un langage intermédiaire entre des outils de compilation de haut niveau et des outils de bas niveau comme des optimiseurs. Les fonctionnalités qui ont été changées ou omises comparé au C, comme les fonctions variadiques, les pointeurs et les parties « avancées » du système de types, auraient entravé les fonctionnalités essentielles de C--, telles que la récursion terminale ou la facilité avec laquelle les outils de génération de code peuvent produire du code.

3. Grammaire améliorée

Pour ce projet, nous intégrons les caractères comme amélioration à la grammaire donnée.

Les points exacts que nous ajoutons sont :

- La déclaration d'une variable de type **car**.
- Le type de retour d'une fonction est du type **car**.
- Les paramètres d'une fonction peuvent être du type **car**.
- La déclaration d'une table de **car**.

Après l'ajout du nécessaire, nous obtenons la grammaire améliorée du langage C--, est la suivante :

<Programme> : <liste-declarations> <liste-fonctions>

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

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

<declaration> : **int** <liste-declarateurs> ; | **car** <liste-declarateurs> ;

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

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

<fonction> : <type> **identificateur** (<liste-parms>) { <liste-declarations> <liste-instructions> }

<type> : **void** | **int** | **car**

<liste-parms> : <liste-parms> , <parm> | **epsilon**

<parm> : **int identificateur** | **car 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>]

<expression> : (<expression>) | <expression> <binary-op> <expression> | - <expression> |

<variable> | **identificateur** (<liste-expressions>) | **constante**

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

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

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

<binary-rel> : && | ||

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

<char> : '<caractere>'

<caractere> : a | ... | z | A | ... | Z | + | - | * | ? | ... | !

<constante> : <chiffre> <constante> | <chiffre>

<chiffre> : 0 | .. | 9

<lettre> : a | .. | z | A | .. | Z

Chapitre II :

Analyse Lexical

1. L'alphabet

D'après la grammaire améliorée, nous précisons l'alphabet suivant :

$$\Sigma = \{a-z \ A-Z \ 0-9 \ ; \ , \ ' \ (\) \ { \ } \ [\] \ = \ < \ > \ \& \ | \ - \ + \ * \ / \ ! \} ;$$

1. Les unités lexicales

| Unités lexicales | Lexèmes / exp | Attribut | Modèle |
|------------------|---------------|------------|---|
| MOTCLE | Voir modèle | (attribut) | int car void for while if then else return lire ecrire |
| PV | « ; » | - | ; |
| VIR | « , » | - | , |
| CRO | « [» | - | [|
| CRF | «] » | - |] |
| ACO | « { » | - | { |
| ACF | « } » | - | } |
| PO | « (» | - | (|
| PF | «) » | - |) |
| MOINS | « - » | - | - |
| NEG | « ! » | - | ! |
| PLUS | « + » | - | + |
| DIFF | « != » | - | != |
| MULT | « * » | - | * |
| DIV | « / » | - | / |
| INF | « < » | - | < |
| SUP | « > » | - | > |
| SUPEG | « >= » | - | >= |
| INFEG | « <= » | - | <= |
| EGAL | « == » | - | == |
| AFF | « = » | - | = |
| IDENT | « score » | (attribut) | (a ... z A ... Z)(a ... z A ... Z 0 .. 9)* |
| CONST | « 5 » | (attribut) | (1-9)(0-9)* |
| ET | « && » | - | && |
| OU | « » | - | |
| CAR | « 'a' » | (attribut) | '(a ... z A ... Z + - * ! ... ?)' |

2. Structure de données

- Nous avons les unités lexicales par énumération :

```
enum UniteLexical { CAR, ERROR, PV,VIR, CRO, CRF,ACO,ACF, PO,PF,
                  MOINS,NOT, PLUS, MULT, DIV,INF,SUP, SUPEG,INFEG,
                  AFF,IDENT,CONST, ET, OU, EGAL, DIFF, MOTCLE
};
```

Figure 1 : Unités lexicales

- La fonction uniteSuivante () retourne :

```
typedef struct {
    UniteLexical UL;
    char *lexeme;
} TUnite;
```

Figure 2 : Type de retour uniteSuivante()

- Les mots clés :

```
vector<string> motCles = {"int", "then", "return", "void", "ecrire",
                        "else", "for", "while", "if", "lire"};
```

Figure 3 : Les mots clés

- L'hachage :

```
int hashage(const string& word) {
    int hash_value = 0;
    const int prime = 31;

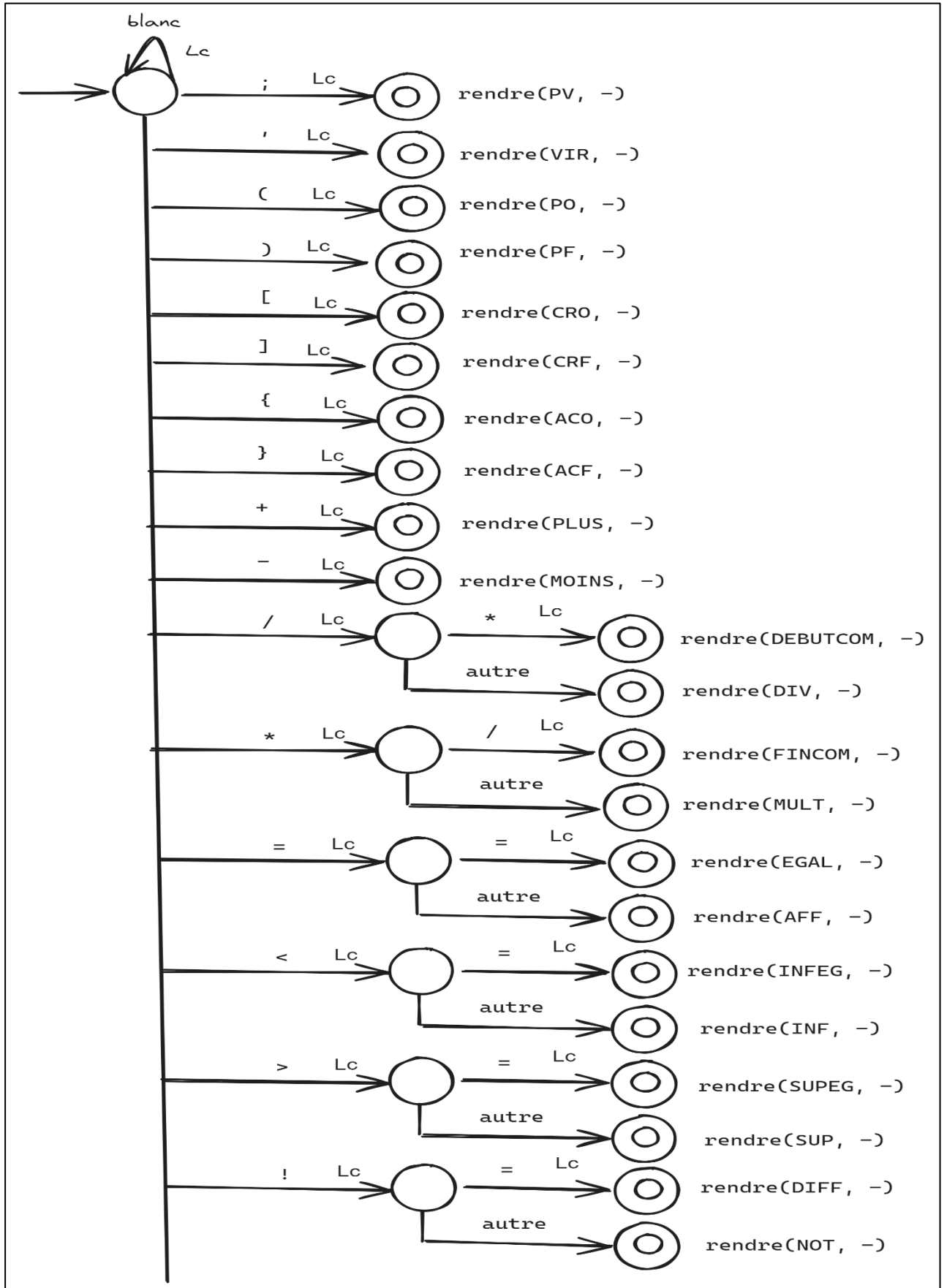
    for (char c : word) {
        hash_value = (hash_value * prime) + static_cast<int>(c);
    }
    hash_value = hash_value % 100;
    if (hash_value > 50) {
        hash_value -= 50;
    }
    return hash_value;
}
```

Figure 4 : Fonction de hashage

```
vector<string> hashed_table(100, "null");
void mot_cles(){
    for (const auto& mot : motCles) {
        int hash_value = hashage(mot);
        hashed_table[hash_value] = mot;
    }
}
```

Figure 5 : Remplir le tableau des mots clés

1. Diagramme d'état



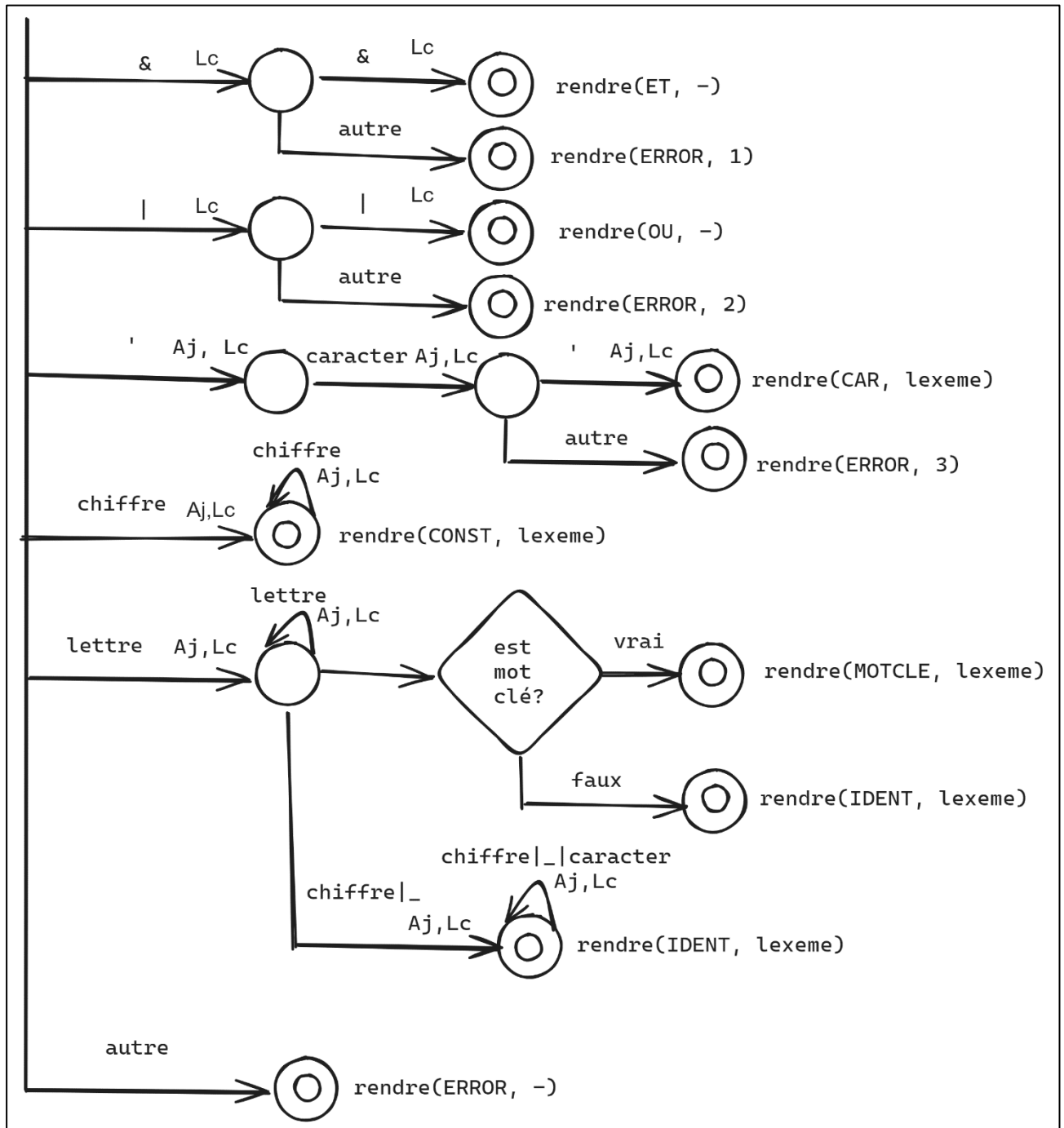


Figure 6: Diagramme d'état

5. Codage

Les programmes qui seront compilés par notre compilateur vont être des fichiers. En effet, la notion de fichier nous donne la possibilité de parcourir un fichier .txt par un curseur, de positionner le curseur où nous voulons.

```
/// ouvrir le fichier source
ifstream file("code.txt");
```

Figure 7 : Ouverture du fichier

En outre, afin de savoir si le curseur à arriver à la fin du fichier ou non on test si le caractère lu est EOF (End Of File). En fait, nous ouvrons le fichier qui contient le code avec le privilège de le lire.

```
int main()
{
    if (!file.is_open())
    {
        cerr << "Erreur en ouverture du fichier" << endl;
        return 1;
    }

    carCourant = lireCar();

    int i = 1;
    while (carCourant != EOF)
    {
        TUnite x = uniteSuivante();
        cout << i << " ) " << x.lexeme << " : " ; affiche(x.UL);
        i++;
    }
    file.close();
}
```

Figure 8 : La fonction main et gestion de la fin du fichier

Les fonctions utiliser dans la fonctions uniteSuivante() :

```
int estBlanc(char c){
    return c == ' ' || c == '\t' || c == '\n';
}

int estLettre(char c){
    return ('A' <= c && c <= 'Z') || ('a' <= c && c <= 'z');
}

int estCar(char c){
    return 0 <= c && c <= 255;
}

int estChiffre(char c){
    return '0' <= c && c <= '9';
}

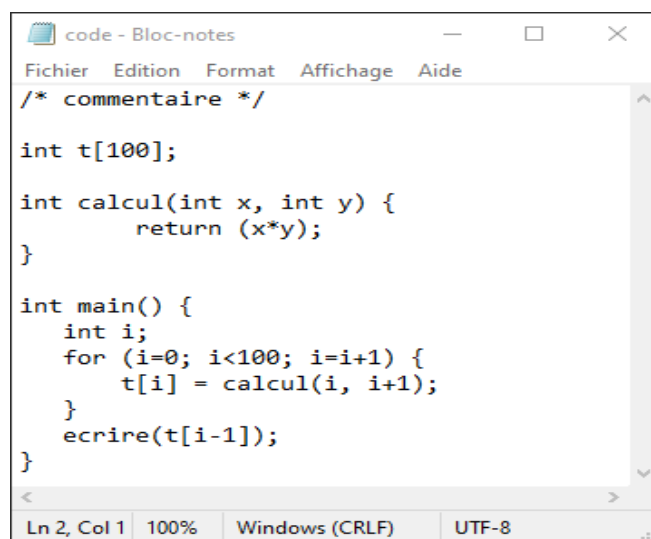
char lireCar() {
    return file.get();
}

void delireCar(char c){
    file.putback(c);
}
```

Figure 9 : Les fonctions utilisées en uniteSuivante()

6. Test

- Code correct



```
code - Bloc-notes
Fichier Edition Format Affichage Aide
/* commentaire */

int t[100];

int calcul(int x, int y) {
    return (x*y);
}

int main() {
    int i;
    for (i=0; i<100; i=i+1) {
        t[i] = calcul(i, i+1);
    }
    ecrire(t[i-1]);
}
```

Ln 2, Col 1 | 100% | Windows (CRLF) | UTF-8

Figure 10 : Code de test

```

1) int : MOTCLE
2) t : IDENT
3) [ : Croche ouvrante
4) 100 : CONST
5) ] : Crochee fermante
6) ; : Point virgule
7) int : MOTCLE
8) calcul : IDENT
9) ( : Parenthese ouvrante
10) int : MOTCLE
11) x : IDENT
12) , : Virgule
13) int : MOTCLE
14) y : IDENT
15) ) : parenthese fermante
16) { : Accollade ouvrante
17) return : MOTCLE
18) ( : Parenthese ouvrante
19) x : IDENT
20) * : MULT
21) y : IDENT
22) ) : parenthese fermante
23) ; : Point virgule
24) } : Accollade fermante
25) int : MOTCLE
26) main : IDENT
27) ( : Parenthese ouvrante
28) ) : parenthese fermante
29) { : Accollade ouvrante
30) int : MOTCLE

```

```

30) int : MOTCLE
31) i : IDENT
32) ; : Point virgule
33) for : MOTCLE
34) ( : Parenthese ouvrante
35) i : IDENT
36) = : AFF
37) 0 : CONST
38) ; : Point virgule
39) i : IDENT
40) < : INF
41) 100 : CONST
42) ; : Point virgule
43) i : IDENT
44) = : AFF
45) i : IDENT
46) + : PLUS
47) 1 : CONST
48) ) : parenthese fermante
49) { : Accollade ouvrante
50) t : IDENT
51) [ : Croche ouvrante
52) i : IDENT
53) ] : Crochee fermante
54) = : AFF
55) calcul : IDENT
56) ( : Parenthese ouvrante
57) i : IDENT
58) , : Virgule
59) i : IDENT

```

```

59) i : IDENT
60) + : PLUS
61) 1 : CONST
62) ) : parenthese fermante
63) ; : Point virgule
64) } : Accollade fermante
65) ecrire : MOTCLE
66) ( : Parenthese ouvrante
67) t : IDENT
68) [ : Croche ouvrante
69) i : IDENT
70) - : MOINS
71) 1 : CONST
72) ] : Crochee fermante
73) ) : parenthese fermante
74) ; : Point virgule
75) } : Accollade fermante

```

```

code_erreur - Bloc-notes
Fichier Edition Format Affichage Aide

int i, j;
char c;
i = 3;
j = 5;
c = 's';

if ( i >= 2 & j < 9 ){
    ecrire( i / j );
}
if ( i != 3 | j > 10 ){
    ecrire( i + j );
}

Ln 1, Col 1 100% Windows (CRLF) UTF-8

```

Figure 11 : Résultat du test

Figure 12 : Code erroné



| | |
|--------------------------------------|-------------------------------------|
| 1) int : MOTCLE | 31)) : parenthese fermante |
| 2) i : IDENT | 32) { : Accollade ouvrante |
| 3) , : Virgule | 33) écrire : MOTCLE |
| 4) j : IDENT | 34) (: Parenthese ouvrante |
| 5) ; : Point virgule | 35) i : IDENT |
| 6) car : IDENT | 36) / : DIV |
| 7) c : IDENT | 37) j : IDENT |
| 8) ; : Point virgule | 38)) : parenthese fermante |
| 9) i : IDENT | 39) ; : Point virgule |
| 10) = : AFF | 40) } : Accollade fermante |
| 11) 3 : CONST | 41) if : MOTCLE |
| 12) ; : Point virgule | 42) (: Parenthese ouvrante |
| 13) j : IDENT | 43) i : IDENT |
| 14) = : AFF | 44) != : DIFF |
| 15) 5 : CONST | 45) 3 : CONST |
| 16) ; : Point virgule | 46) Erreur ' ' est attendu : Erreur |
| 17) c : IDENT | 47) j : IDENT |
| 18) = : AFF | 48) > : SUP |
| 19) Erreur un ' est attendu : Erreur | 49) 10 : CONST |
| 20) s : IDENT | 50)) : parenthese fermante |
| 21) ; : Point virgule | 51) { : Accollade ouvrante |
| 22) if : MOTCLE | 52) écrire : MOTCLE |
| 23) (: Parenthese ouvrante | 53) (: Parenthese ouvrante |
| 24) i : IDENT | 54) i : IDENT |
| 25) >= : SUPEG | 55) + : PLUS |
| 26) 2 : CONST | 56) j : IDENT |
| 27) Erreur '&' est attendu : Erreur | 57)) : parenthese fermante |
| 28) j : IDENT | 58) ; : Point virgule |
| 29) < : INF | 59) } : Accollade fermante |
| 30) 9 : CONST | |

Figure 13 : Résultat de code erroné



Chapitre III :

Analyse Syntaxique

I. Grammaire LL (1) :

2. Grammaire améliorée LL (1)

Nous soumettons la grammaire améliorée sous les différentes étapes, nous obtenons :

<Programme> : int identificateur <Programme> | car identificateur <Programme> | void identificateur (<liste-parms>){ <liste-declarations> <liste-instructions> } <liste-fonctions> | epsilon

<Programme'> : <declarateur'> <liste-declarateurs'> ; <Programme> | (<liste-parms>){ <liste-declarations> <liste-instructions>} <liste-fonctions>

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

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

<declaration> : int <liste-declarateurs> ; | car < liste-declarateurs> ;

<liste-declarateurs> : <declarateur> <liste-declarateurs'>

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

<declarateur> : identificateur <declarateur'>

<declarateur'> : [<constante>] | epsilon

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

<type> : void | int | car

<liste-parms> : <parm> <liste-parms'> | epsilon

<liste-parms'> : , <parm> <liste-parms'> | epsilon

<parm> : int identificateur | car identificateur

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

<instruction> : <iteration> | <selection> | <saut> | <bloc> | identificateur <appel/affectation>

<appel/affectation> : [<expression>] = <expression>; | = <expression>; | (<liste-expressions>);

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

<selection> : **if**(<condition>) { <instruction> } <else'>

<else'> : **else** { <instruction> } | **epsilon**

<saut> : **return** <saut'>

<saut'> : <expression> ; | ;

<affectation> : <variable> = <expression>

<bloc> : {<liste-instructions>}

<variable> : **identificateur** <variable'>

<variable'> : [<expression>] | **epsilon**

<liste-expressions> : <expression> <liste-expressions'> | **epsilon**

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

<expression> : <terme> <expression_prime> | - <expression>

<expression_prime> : + <terme> <expression_prime> | - <terme> <expression_prime> | **epsilon**

<terme> : <facteur> <terme_prime>

<terme_prime> : * <facteur> <terme_prime> | / <facteur> <terme_prime> | **epsilon**

<facteur> : **identificateur** <appel/variable> | <constante> | <char> | (<expression>)

<condition> : !(<condition>) <suite_condition> | <expression> <condition'>

<condition'> : <binary-comp> <expression> <suite_condition> | **epsilon**

<suite_condition> : <binary-rel> <condition> | **epsilon**

<appel/variable> : (<liste-expressions>) | <variable'>

<binary-rel> : **&&** | **||**

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

<constante> : (1|..|9)(0|...|9)*

<char> : 'caractere'



2. Calcule des premiers et suivants

| Dérivation | Premiers | Suivants |
|------------------------------------|---|---|
| Programme | int, car, void | \$ |
| Programme' | [, , , ; ,) | \$ |
| <liste-declarations> | int , car , epsilon | for , while , if , return , ident , { , } |
| <liste_fonctions> | Int, car, ε | \$ |
| <declaration > | int , car , void , epsilon | for , while , if , return , ident , { , } |
| <liste-declarateur> | ident | ; |
| <liste-declarateur'> | , , epsilon | ; |
| < declarateur> | ident | , , ; |
| < declarateur'> | [, epsilon | , , ; |
| < fonction> | int , car , void | int , car , void , \$ |
| < type> | int , car , void | ident |
| < liste-parms> | int , car , epsilon |) |
| < liste-parms'> | , , epsilon |) |
| < parms> | int , car | , ,) |
| < liste-instruction > | for , while , if , return, ident, { | } |
| < instruction > | for , while , if , return, ident, { | for , while , if , return, ident, { , } |
| < appel-affectation > | for , while , if , return, ident, { , } |] , = , (|
| < iteration > | for , while | for , while , if , return, ident, { , } |
| < selection> | if | }, if, while, for, return, {, ident |
| < else'> | else , epsilon | for , while , if , return, ident, { , } |
| < saut > | return | for , while , if , return, ident, { , } |

| | | |
|---------------------|--------------------------------------|--|
| < saut ‘ > | ;; ident, char, const, (| for , while , if , return, ident, { , } |
| <bloc > | { | for , while , if , return, ident, { , } |
| < variable> | ident | (, [=, <binary-comp> , &&, , ; ,) ,<binary-op>, ; |
| < variable ‘> | [, epsilon | =, <binary-comp> , &&, , ; ,) ,<binary-op> |
| <liste-expressions> | ident , - , const , car , epsilon |) |
| <expressions> | ident , - , const , car , (|] , ; , , ,) , <binary-comp> , && , |
| <expressions’> | + , - , epsilon |] , ; , , ,) , <binary-comp> , && , |
| <terme> | | |
| <terme’> | * , / , epsilon |] , ; , , ,) , <binary-comp> , && , |
| <facteur> | ident , const , car |] , ; , , ,) , <binary-comp> , && , , + , - , * , / |
| <binary-rel> | &&, | |
| <binary-comp> | <, >, >=, <=, ==, != | |
| <condition > | ! , ident , const , char , (, - |) , ; |
| <condition’ > | | |
| <suite-condition > | epsilon , && , |) , ; |
| <appel/variable> | (, [, ε |] , ; , , ,) , <binary-comp> , && , , + , - , * , / |

3. Justification du LL (1)

1) Règles pour <programme> :

<Programme> : int identificateur <Programme’> | car identificateur <Programme’> | void
identificateur (<liste-parms>){ <liste-declarations> <liste-instructions> } <liste-fonctions> | ε

- $\text{Premier}(\langle \text{programme} \rangle) = \{ \text{int}, \text{car}, \text{void} \}$
 - $\text{Suivant}(\langle \text{programme} \rangle) = \{ \$ \}$
 - ✓ $\text{Premier}(\text{int identificateur} \langle \text{Programme}' \rangle) \cap \text{Premier}(\text{car identificateur} \langle \text{Programme}' \rangle) \cap \text{Premier}(\text{void identificateur} (\langle \text{liste-parms} \rangle) \{ \langle \text{liste-declarations} \rangle \langle \text{liste-instructions} \rangle \} \langle \text{liste-fonctions} \rangle) \cap \text{Premier}(\epsilon) = \emptyset$
 - ✓ $\text{Suivant}(\langle \text{programme} \rangle) \cap \text{Premier}(\langle \text{programme} \rangle) = \emptyset$
- $\Rightarrow \text{LL}(1)$**

2) Règles pour $\langle \text{programme}' \rangle$:

$\langle \text{Programme}' \rangle : \langle \text{declarateur}' \rangle \langle \text{liste-declarateurs}' \rangle ; \langle \text{Programme}' \rangle | (\langle \text{liste-parms} \rangle) \{ \langle \text{liste-declarations} \rangle \langle \text{liste-instructions} \rangle \} \langle \text{liste-fonctions} \rangle$

- $\text{Premier}(\langle \text{programme}' \rangle) = \{ [, ;, (, \epsilon \}$
 - $\text{Suivant}(\langle \text{programme}' \rangle) = \{ \$ \}$
 - ✓ $\text{Premier}(\langle \text{declarateur}' \rangle \langle \text{liste-declarateurs}' \rangle ; \langle \text{Programme}' \rangle) \cap \text{Premier}((\langle \text{liste-parms} \rangle) \{ \langle \text{liste-declarations} \rangle \langle \text{liste-instructions} \rangle \} \langle \text{liste-fonctions} \rangle) = \emptyset$
 - ✓ Aucun ne se dérive en epsilon
- $\Rightarrow \text{LL}(1)$**

3) Règles pour $\langle \text{liste-declarations} \rangle$:

$\langle \text{liste-declarations} \rangle : \langle \text{declaration} \rangle \langle \text{liste-declarations} \rangle | \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-declarations} \rangle) = \{ \text{for}, \text{while}, \text{if}, \text{return}, \text{ident}, \{, \} \}$
 - $\text{Premier}(\langle \text{liste-declarations} \rangle) = \{ \text{int}, \text{car}, \text{epsilon} \}$
 - ✓ $\text{Premier}(\langle \text{declaration} \rangle \langle \text{liste-declarations} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
 - ✓ $\text{Suivant}(\langle \text{liste-declarations} \rangle) \cap \text{Premier}(\langle \text{liste-declarations} \rangle) = \emptyset$
- $\Rightarrow \text{LL}(1)$**

4) Règles pour $\langle \text{liste-fonctions} \rangle$:

$\langle \text{liste-fonctions} \rangle : \langle \text{fonction} \rangle \langle \text{liste-fonctions} \rangle | \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-fonctions} \rangle) = \{ \$ \}$
 - $\text{Premier}(\langle \text{liste-fonctions} \rangle) = \{ \text{int}, \text{car}, \text{void}, \text{epsilon} \}$
 - ✓ $\text{Premier}(\langle \text{fonction} \rangle \langle \text{liste-fonctions} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
 - ✓ $\text{Suivant}(\langle \text{liste-fonctions} \rangle) \cap \text{Premier}(\langle \text{liste-fonctions} \rangle) = \emptyset$
- $\Rightarrow \text{LL}(1)$**

5) Règles pour <declaration> :

<declaration> : int <liste-declarateurs> ; | car < liste-declarateurs> ;

- Suivant(<declarations>) = { for , while , if , return , ident , { , } }
 - Premier (<declarations>) = { int , car , void , epsilon }
 - ✓ Premier (int <liste-declarateurs> ;) \cap Premier (car < liste-declarateurs> ;) = \emptyset
 - ✓ Aucune ne se dérive en epsilon
- => LL(1)**

6) Règles pour <liste-declarateur> :

<liste-declarateurs> : <declarateur> <liste-declarateurs'>

- Suivant(<liste-declarateur>) = { ; }
 - Premier (<liste-declarateur>) = { ident }
 - ✓ <declarateur> <liste-declarateurs'> ne se dérive en epsilon
- => LL(1)**

7) Règles pour <liste-declarateur'> :

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

- Suivant(<liste-declarateur'>) = { ; }
 - Premier (<liste-declarateur'>) = { , , epsilon }
 - ✓ Premier (, <declarateur> <liste-declarateurs'>) \cap Premier (epsilon) = \emptyset
 - ✓ Suivant(<liste-declarateur'>) \cap Premier (<liste-declarateur'>) = \emptyset
- => LL(1)**

8) Règles pour < declarateur> :

<declarateur> : identificateur <declarateur'>

- Suivant(<declarateur>) = { , , ; }
 - Premier (<declarateur>) = { ident }
 - ✓ identificateur <declarateur'> ne se dérive en epsilon
- => LL(1)**

9) Règles pour < declarateur'> :

<declarateur'> : [constante] | epsilon

- $\text{Suivant}(\langle \text{declarateur}' \rangle) = \{ , , ; \}$
- $\text{Premier}(\langle \text{declarateur}' \rangle) = \{ [, \text{epsilon} \}$
- ✓ $\text{Premier}(\langle [\text{constante}] \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{declarateur}' \rangle) \cap \text{Premier}(\langle \text{declarateur}' \rangle) = \emptyset$

$\Rightarrow \text{LL}(1)$

10) Règles pour $\langle \text{fonction} \rangle$:

$\langle \text{fonction} \rangle : \langle \text{type} \rangle \text{ identificateur } (\langle \text{liste-parms} \rangle) \{ \langle \text{liste-declarations} \rangle \langle \text{liste-instructions} \rangle \}$

- $\text{Suivant}(\langle \text{fonction} \rangle) = \{ \text{int} , \text{car} , \text{void} , \$ \}$
- $\text{Premier}(\langle \text{fonction} \rangle) = \{ \text{int} , \text{car} , \text{void} \}$
- ✓ $\langle \text{type} \rangle \text{ identificateur } (\langle \text{liste-parms} \rangle) \{ \langle \text{liste-declarations} \rangle \langle \text{liste-instructions} \rangle \}$ ne se derive pas de epsilon

$\Rightarrow \text{LL}(1)$

11) Règles pour $\langle \text{type} \rangle$:

$\langle \text{type} \rangle : \text{void} \mid \text{int} \mid \text{car}$

- $\text{Suivant}(\langle \text{type} \rangle) = \{ \text{ident} \}$
- $\text{Premier}(\langle \text{type} \rangle) = \{ \text{int} , \text{car} , \text{void} \}$
- ✓ $\text{Premier}(\text{int}) \cap \text{Premier}(\text{car}) \cap \text{Premier}(\text{void}) = \emptyset$
- ✓ ni **int** ni **car** ni **void** annuable

$\Rightarrow \text{LL}(1)$

12) Règles pour $\langle \text{liste-parms} \rangle$:

$\langle \text{liste-parms} \rangle : \langle \text{parm} \rangle \langle \text{liste-parms}' \rangle \mid \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-parms} \rangle) = \{) \}$
- $\text{Premier}(\langle \text{liste-parms} \rangle) = \{ \text{int} , \text{car} , \text{epsilon} \}$
- ✓ $\text{Premier}(\langle \text{parm} \rangle \langle \text{liste-parms}' \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{liste-parms} \rangle) \cap \text{Premier}(\langle \text{liste-parms} \rangle) = \emptyset$

$\Rightarrow \text{LL}(1)$

13) Règles pour $\langle \text{liste-parms}' \rangle$:

$\langle \text{liste-parms} \rangle : , \langle \text{parm} \rangle \langle \text{liste-parms} \rangle \mid \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-parms} \rangle) = \{ , \}$
- $\text{Premier}(\langle \text{liste-parms} \rangle) = \{ , , \text{epsilon} \}$
- ✓ $\text{Premier}(\langle \text{parm} \rangle \langle \text{liste-parms} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{liste-parms} \rangle) \cap \text{Premier}(\langle \text{liste-parms} \rangle) = \emptyset$

$\Rightarrow \text{LL(1)}$

14) Règles pour $\langle \text{parms} \rangle$:

$\langle \text{parm} \rangle : \text{int identificateur} \mid \text{car identificateur}$

- $\text{Suivant}(\langle \text{parms} \rangle) = \{ , ,) \}$
- $\text{Premier}(\langle \text{parms} \rangle) = \{ \text{int} , \text{car} \}$
- ✓ $\text{Premier}(\text{int identificateur}) \cap \text{Premier}(\text{car identificateur}) = \emptyset$
- ✓ Aucun des deux ne se derive en epsilon

$\Rightarrow \text{LL(1)}$

15) Règles pour $\langle \text{liste-instruction} \rangle$:

$\langle \text{liste-instructions} \rangle : \langle \text{instruction} \rangle \langle \text{liste-instructions} \rangle \mid \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-instruction} \rangle) = \{ \} \}$
- $\text{Premier}(\langle \text{liste-instruction} \rangle) = \{ \text{for} , \text{while} , \text{if} , \text{return} , \text{ident} , \{ \} \}$
- ✓ $\text{Premier}(\langle \text{instruction} \rangle \langle \text{liste-instructions} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{liste-instruction} \rangle) \cap \text{Premier}(\langle \text{liste-instruction} \rangle) = \emptyset$

$\Rightarrow \text{LL(1)}$

16) Règles pour $\langle \text{instruction} \rangle$:

$\langle \text{instruction} \rangle : \langle \text{iteration} \rangle \mid \langle \text{selection} \rangle \mid \langle \text{saut} \rangle \mid \langle \text{bloc} \rangle \mid \text{identificateur} \langle \text{appel/affectation} \rangle$

- $\text{Suivant}(\langle \text{instruction} \rangle) = \{ \text{for} , \text{while} , \text{if} , \text{return} , \text{ident} , \{ , \} \}$
- $\text{Premier}(\langle \text{instruction} \rangle) = \{ \text{for} , \text{while} , \text{if} , \text{return} , \text{ident} , \{ \} \}$
- ✓ $\text{Premier}(\langle \text{iteration} \rangle) \cap \text{Premier}(\langle \text{selection} \rangle) \cap \text{Premier}(\langle \text{bloc} \rangle) \cap \text{Premier}(\langle \text{saut} \rangle) \cap \text{Premier}(\text{identificateur} \langle \text{appel/affectation} \rangle) = \emptyset$
- ✓ Aucune n'est annulable

$\Rightarrow LL(1)$

17) Règles pour $\langle \text{appel-affectation} \rangle$:

$\langle \text{appel/affectation} \rangle : [\langle \text{expression} \rangle] = \langle \text{expression} \rangle ; | = \langle \text{expression} \rangle ; | (\langle \text{liste-expressions} \rangle) ;$

- $\text{Suivant}(\langle \text{appel-affectation} \rangle) = \{ \text{for}, \text{while}, \text{if}, \text{return}, \text{ident}, \{ , \} \}$
- $\text{Premier}(\langle \text{appel-affectation} \rangle) = \{], =, (\}$
- ✓ $\text{Premier}([\langle \text{expression} \rangle] = \langle \text{expression} \rangle ;) \cap \text{Premier}(= \langle \text{expression} \rangle ;) \cap \text{Premier}((\langle \text{liste-expressions} \rangle) ;) = \emptyset$
- ✓ Aucune n'est annulable

$\Rightarrow LL(1)$

18) Règles pour $\langle \text{iteration} \rangle$:

$\langle \text{iteration} \rangle : \text{for}(\langle \text{affectation} \rangle ; \langle \text{condition} \rangle ; \langle \text{affectation} \rangle) \langle \text{instruction} \rangle | \text{while}(\langle \text{condition} \rangle) \langle \text{instruction} \rangle$

- $\text{Suivant}(\langle \text{iteration} \rangle) = \{ \text{for}, \text{while}, \text{if}, \text{return}, \text{ident}, \{ , \} \}$
- $\text{Premier}(\langle \text{iteration} \rangle) = \{ \text{for}, \text{while} \}$
- ✓ $\text{Premier}(\text{for}(\langle \text{affectation} \rangle ; \langle \text{condition} \rangle ; \langle \text{affectation} \rangle) \langle \text{instruction} \rangle) \cap \text{Premier}(\text{while}(\langle \text{condition} \rangle) \langle \text{instruction} \rangle) = \emptyset$
- ✓ Aucune n'est annulable

$\Rightarrow LL(1)$

19) Règles pour $\langle \text{selection} \rangle$:

$\langle \text{selection} \rangle : \text{if}(\langle \text{condition} \rangle) \{ \langle \text{instruction} \rangle \} \langle \text{else}' \rangle$

- c'est bon

20) Règles pour $\langle \text{else}' \rangle$:

- $\text{Suivant}(\langle \text{else}' \rangle) = \{ \text{for}, \text{while}, \text{if}, \text{return}, \text{ident}, \{ , \} \}$
- $\text{Premier}(\langle \text{else}' \rangle) = \{ \text{else}, \text{epsilon} \}$
- ✓ $\text{Premier}(\text{else} \{ \langle \text{instruction} \rangle \}) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{else}' \rangle) \cap \text{Premier}(\langle \text{else}' \rangle) = \emptyset$

$\Rightarrow LL(1)$

21) Règles pour $\langle \text{saut} \rangle$:

$\langle \text{saut} \rangle : \text{return } \langle \text{saut}' \rangle$

- **c'est bon.**

$\Rightarrow \text{LL}(1)$

22) Règles pour $\langle \text{saut} ' \rangle$:

$\langle \text{saut}' \rangle : \langle \text{expression} \rangle ; | ;$

✓ Premier ($\langle \text{expression} \rangle ;$) \cap Premier ($;$) = \emptyset

✓ Aucune n'est annulable

$\Rightarrow \text{LL}(1)$

23) Règles pour $\langle \text{affectation} \rangle$:

- $\langle \text{affectation} \rangle : \langle \text{variable} \rangle = \langle \text{expression} \rangle ;$

- **c'est bon.**

$\Rightarrow \text{LL}(1)$

24) Règles pour $\langle \text{bloc} \rangle$:

- $\langle \text{bloc} \rangle : \{ \langle \text{liste-instructions} \rangle \}$

- **c'est bon.**

$\Rightarrow \text{LL}(1)$

25) Règles pour $\langle \text{variable} \rangle$:

- $\langle \text{variable} \rangle : \text{identificateur } \langle \text{variable}' \rangle$

- **c'est bon.**

$\Rightarrow \text{LL}(1)$

26) Règles pour $\langle \text{variable} ' \rangle$:

$\langle \text{variable}' \rangle : [\langle \text{expression} \rangle] \mid \text{epsilon}$

➤ Suivant($\langle \text{variable} ' \rangle$) = $\{ =, \langle \text{binary-comp} \rangle, \langle \text{binary-rel} \rangle, ;, , \rangle, \langle \text{binary-op} \rangle \}$

➤ Premier ($\langle \text{variable} ' \rangle$) = $\{ [, \text{epsilon} \}$

✓ Premier ($[\langle \text{expression} \rangle]$) \cap Premier (epsilon) = \emptyset

✓ Suivant ($\langle \text{variable} ' \rangle$) \cap Premier ($\langle \text{variable} ' \rangle$) = \emptyset

$\Rightarrow \text{LL}(1)$

27) Règles pour $\langle \text{liste-expressions} \rangle$:

$\langle \text{liste-expressions} \rangle : \langle \text{expression} \rangle \langle \text{liste-expressions}' \rangle \mid \text{epsilon}$

- $\text{Suivant}(\langle \text{liste-expressions} \rangle) = \{ \} \}$
 - $\text{Premier}(\langle \text{liste-expressions} \rangle) = \{ \text{ident}, -, \text{const}, \text{car}, \text{epsilon} \}$
 - ✓ $\text{Premier}(\langle \text{expression} \rangle \langle \text{liste-expressions}' \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
 - ✓ $\text{Suivant}(\langle \text{liste-expressions} \rangle) \cap \text{Premier}(\langle \text{liste-expressions} \rangle) = \emptyset$
- $\Rightarrow \text{LL}(1)$**

28) Règles pour $\langle \text{expressions} \rangle$:

$\langle \text{expression} \rangle : \langle \text{terme} \rangle \langle \text{expression_prime} \rangle \mid - \langle \text{expression} \rangle$

- $\text{Suivant}(\langle \text{expressions} \rangle) = \{], ; , , , \rangle, \langle \text{binary-comp} \rangle, \&\&, || \}$
- $\text{Premier}(\langle \text{expressions} \rangle) = \{ \text{ident}, -, \text{const}, \text{car}, (\}$
- ✓ $\text{Premier}(\langle \text{terme} \rangle \langle \text{expression_prime} \rangle) \cap \text{Premier}(- \langle \text{expression} \rangle) = \emptyset$
- ✓ aucune des deux n'est annulable

$\Rightarrow \text{LL}(1)$

29) Règles pour $\langle \text{expressions}' \rangle$:

- $\text{Suivant}(\langle \text{expressions}' \rangle) = \{], ; , , , \rangle, \langle \text{binary-comp} \rangle, \&\&, || \}$
- $\text{Premier}(\langle \text{expressions}' \rangle) = \{ +, -, \text{epsilon} \}$
- ✓ $\text{Premier}(+ \langle \text{terme} \rangle \langle \text{expression_prime} \rangle) \cap \text{Premier}(- \langle \text{terme} \rangle \langle \text{expression_prime} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{expressions}' \rangle) \cap \text{Premier}(\langle \text{expressions}' \rangle) = \emptyset$

$\Rightarrow \text{LL}(1)$

30) Règles pour $\langle \text{terme} \rangle$:

- $\langle \text{terme} \rangle : \langle \text{facteur} \rangle \langle \text{terme_prime} \rangle$
- **c'est bon**

$\Rightarrow \text{LL}(1)$

31) Règles pour $\langle \text{terme}' \rangle$:

- $\text{Suivant}(\langle \text{terme}' \rangle) = \{], ; , , , \rangle, \langle \text{binary-comp} \rangle, \&\&, || \}$
- $\text{Premier}(\langle \text{terme}' \rangle) = \{ *, / , \text{epsilon} \}$
- ✓ $\text{Premier}(* \langle \text{facteur} \rangle \langle \text{terme_prime} \rangle) \cap \text{Premier}(/ \langle \text{facteur} \rangle \langle \text{terme_prime} \rangle) \cap \text{Premier}(\text{epsilon}) = \emptyset$
- ✓ $\text{suivant}(\langle \text{terme}' \rangle) \cap \text{Premier}(\langle \text{terme}' \rangle) = \emptyset$

$\Rightarrow \text{LL}(1)$

32) Règles pour $\langle \text{facteur} \rangle$:

- $\text{Suivant}(\langle \text{facteur} \rangle) = \{], ; , , , \rangle , \langle \text{binary-comp} \rangle , \&\& , || , + , - , * , / \}$
- $\text{Premier}(\langle \text{facteur} \rangle) = \{ \text{ident} , \text{const} , \text{car} \}$
- ✓ $\text{Premier}(\text{identificateur} \langle \text{appel/variable} \rangle) \cap \text{Premier}(\text{const}) \cap \text{Premier}(\text{char}) \cap \text{Premier}(\langle \text{expression} \rangle) = \emptyset$
- ✓ aucune n'est annulable

=> LL(1)

33) Règles pour $\langle \text{binary-rel} \rangle$:

- $\langle \text{binary-rel} \rangle : \&\& | ||$
- c'est bon

=> LL(1)

34) Règles pour $\langle \text{binary-comp} \rangle$:

- $\langle \text{binary-comp} \rangle : < | > | >= | <= | == | !=$
- -c'est bon

=> LL(1)

35) Règles pour $\langle \text{condition} \rangle$:

- $\text{Suivant}(\langle \text{condition} \rangle) = \{ \rangle , ; \}$
- $\text{Premier}(\langle \text{condition} \rangle) = \{ ! , \text{ident} , \text{const} , \text{car} , (, - \}$
- ✓ $\text{Premier}(!(\langle \text{condition} \rangle) \langle \text{suite_condition} \rangle) \cap \text{Premier}(\langle \text{expression} \rangle \langle \text{condition}' \rangle) = \emptyset$
- ✓ Aucune n'est annulable

=> LL(1)

36) Règles pour $\langle \text{condition}' \rangle$:

- $\text{Premier}(\langle \text{binary-comp} \rangle \langle \text{expression} \rangle \langle \text{suite_condition} \rangle) \cap \text{Premier}(\epsilon) = \emptyset$
- $\text{Suivant}(\langle \text{condition} \rangle) \cap \text{Premier}(\langle \text{condition} \rangle) = \emptyset$

=> LL(1)

37) Règles pour $\langle \text{suite-condition} \rangle$:

- $\text{Suivant}(\langle \text{suite-condition} \rangle) = \{ \rangle , ; \}$
- $\text{Premier}(\langle \text{suite-condition} \rangle) = \{ \epsilon , \&\& , || \}$
- ✓ $\text{Premier}(\langle \text{binary-rel} \rangle \langle \text{condition} \rangle) \cap \text{Premier}(\epsilon) = \emptyset$
- ✓ $\text{Suivant}(\langle \text{suite-condition} \rangle) \cap \text{Premier}(\langle \text{suite-condition} \rangle) = \emptyset$

=> LL(1)

38) Règles pour $\langle \text{appel/variable} \rangle$:

- $\text{Suivant}(\langle \text{appel/variable} \rangle) = \{], ; , , ,) , \langle \text{binary-comp} \rangle , \&\& , || , + , - , * , / \}$
- $\text{Premier}(\langle \text{appel/variable} \rangle) = \{ (, [, \epsilon \}$
- ✓ $\text{Premier}(\langle \text{liste-expressions} \rangle) \cap \text{Premier}(\text{variable}') = \emptyset$
- ✓ $\text{Suivant}(\langle \text{suite-condition} \rangle) \cap \text{Premier}(\langle \text{suite-condition} \rangle) = \emptyset$

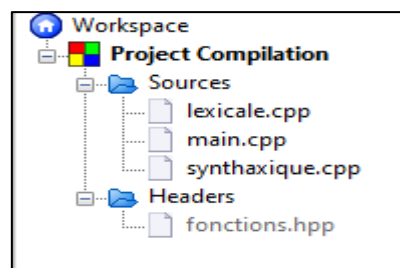
$\Rightarrow \text{LL}(1)$

II. Codage

1. Le code sources du syntaxique

Pour chaque non terminal nous avons une fonction.

Voici la structure de notre projet :



Dans le fichier lexecile.cpp nous avons les fonctions de lexicales, et dans le fichier syntaxique.cpp les fonctions correspondantes aux terminaux. Et dans le fichier header fonctions.hpp nous avons les entêtes des fonctions.

Nous notre programme utilise 3 variables globales :

- ❖ **carCourant** : qui contient le caractère courant, et il est utilisé par la fonction **uniteSuivante()** et initialisé par le syntaxique.
- ❖ **motCourant** : contenant le mot courant et utilisé par les fonctions du syntaxique.

Voici notre fonction main :

```
int main()
{
    mot_cles();
    if (!file.is_open())
    {
        cerr << "Erreur en ouverture du fichier" << endl;
        return 1;
    }

    carCourant = lireCar();
    motCourant = uniteSuivante();
    programme();

    if(compilation){
        cout << "Votre code est correct synthaxiquement" << endl;
    }

    file.close();
    return 0;
}
```

Figure 14 : Fonction main

2. Teste

❖ Le cas d'un code source sans erreurs syntaxiques

```
code - Bloc-notes
Fichier Edition Format Affichage Aide
/* Commentaire Premiere code C++ */
int t[100];

int calcul(int x, int y) {
    return 45;
}

int a(){
    t[0] = 34;
}

Ln 9, C 100% Windows (CRLF) UTF-8
```

```
Votre code est correct synthaxiquement
Process returned -1073741787 (0xC0000025)   executio
Press any key to continue.
```

❖ Le cas d'un code source avec erreurs syntaxiques

```
code_erreur - Bl...
Fichier Edition Format Affichage Aide

int i, j;

int main(){
    int a;
    a = 3;
    if( a <= 3 ){
        return
    }
}
```

```
Erreur: ; attendu.
Process returned 0 (0x0)   execution tim
Press any key to continue.
```




Chapitre IV :

Analyse Sémantique

1. Définition

L'analyse sémantique est une phase cruciale du processus de compilation qui intervient après l'analyse syntaxique et avant la génération de code. Son objectif principal est de vérifier la sémantique du code source, c'est-à-dire son sens et sa cohérence par rapport aux règles du langage de programmation utilisé.

2. Périmètre de l'analyse sémantique

Dans cette partie, nous allons contrôler :

- La non déclaration d'une variable
- La double déclaration d'une variable
- Le contrôle de type entre déclaration et utilisation d'une variable.
- Contrôle d'appel à une fonction
- Le contrôle d'affectation type de retour d'une fonction à une variable.

3. Les choix techniques

Pour coder l'analyse sémantique nous avons implémenter trois classes suivantes :

```
class Symbole {
public:
    string nom;
    Classe classe;
    Type type;
    Symbole(string n, Type t, Classe c) : nom(n), type(t), classe(c) {}
    Symbole(){}
};

class Variable: public Symbole {
public :
    Variable(string n, Type t, Classe c):Symbole(n,t,c){
    }
};

class Fonction: public Symbole {
public :
    vector <Symbole*> variablesLocal;
```

```
vector <Symbole*> parametres;
Fonction(string n, Type t, Classe c):Symbole(n,t,c){
}
Fonction(){}
};
class Tableau: public Symbole {
public:
    int taille;
    Tableau(string n, Type t, Classe c, int ta):Symbole(n,t,c), taille(ta){
    }
};
```

Nous avons assez créé une table des symboles qui contiendra les symboles et données de code source. Ainsi que les méthodes permettant de manipuler les ce tableau des symboles :

```
vector <Symbole*> listesymboles; /// La table des symboles
```

La méthode existe qui vérifie si un symbole existe dans la liste des symboles :

```
bool existe(string s, vector<Symbole*> v)
{
    for (Symbole* symbole : v)
    {
        if (symbole->nom == s)
        {
            return true;
        }
    }
    return false;
}
```

La fonction qui retourne le symbole qui existe dans la table des symboles :

```
Symbole* findSymbol(vector<Symbole*> symbols, string nom)
{
    for (Symbole* symbol : symbols)
    {
        if (symbol->nom == nom)
        {
            return symbol;
        }
    }
}
```

La fonction qui teste si les types des paramètres respects la signature de fonction :

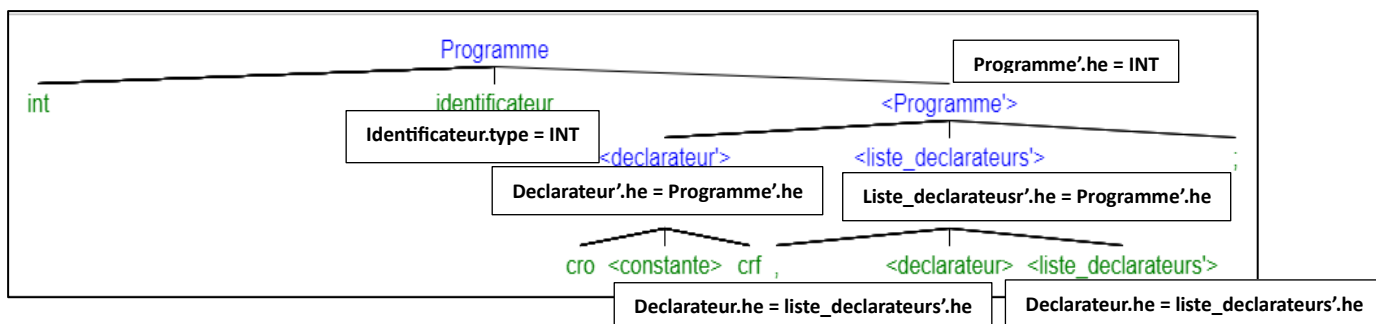
```
bool estTypesCorrects(vector<Symbole*> parametres, vector<Type> liste){
    for (int i = 0; i < liste.size(); i++){
        if (liste[i] != ((parametres[i])>->type)) return false;
    }
    return true;
}
```

4. Les actions sémantiques

Nous définissons les actions sémantiques, précisément les attributs synthétisés et hérités dans les règles de production pour le contrôle de type dans les symboles qui concernent la déclaration.

❖ Les variables globales :

- **<Programme> : int identificateur** {identificateur.type = INT} { Programme'.he = INT}
<Programme'> | car identificateur {identificateur.type = CAR} {Programme'.he = CAR}
<Programme'> | void identificateur { identificateur.type = VOID } (<liste-parms>){ <liste-declarations> <liste-instructions> } <liste-fonctions> | **epsilon**
- **<Programme'> : {declarateur'.he = Programme'.he}** <declarateur'> {liste-declarateurs'.he = Programme'.he} <liste-declarateurs'> ; <Programme> | (<liste-parms>){ <liste-declarations> <liste-instructions>} <liste-fonctions>
- **<declarateur'> : [<constante>] | epsilon**
- **<liste-declarateurs'> : , {declarateur'.he = liste-declarateurs'.he} <declarateur'> { liste-declarateurs'1.he = liste-declarateurs'.he} <liste-declarateurs'> | epsilon**
- **<declarateur'> : identificateur** { identificateur.type = declarateur.he } {declarateurs'.he = declarateur.he} <declarateur'>



❖ Les variables locales :

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

<declaration> : int {liste-declarateurs.he = INT} <liste-declarateurs> ; | car {liste-declarateurs.he = CAR} <liste-declarateurs> ;

<liste-declarateurs> : {declarateur.he = liste-declarateurs.he} <declarateur> {liste-declarateurs'.he = liste-declarateurs.he} <liste-declarateurs'>

<liste-declarateurs'> : , {declarateur.he=liste-declarateurs'.he} <declarateur> {liste_declarateurs'.l.he=liste-declarateurs'.he} <liste-declarateurs'> | epsilon

<declarateur> : **identificateur** {declarateur'.he=declarateur.he} <declarateur'>

<declarateur'> : [constante] | epsilon

❖ Les types des expressions :

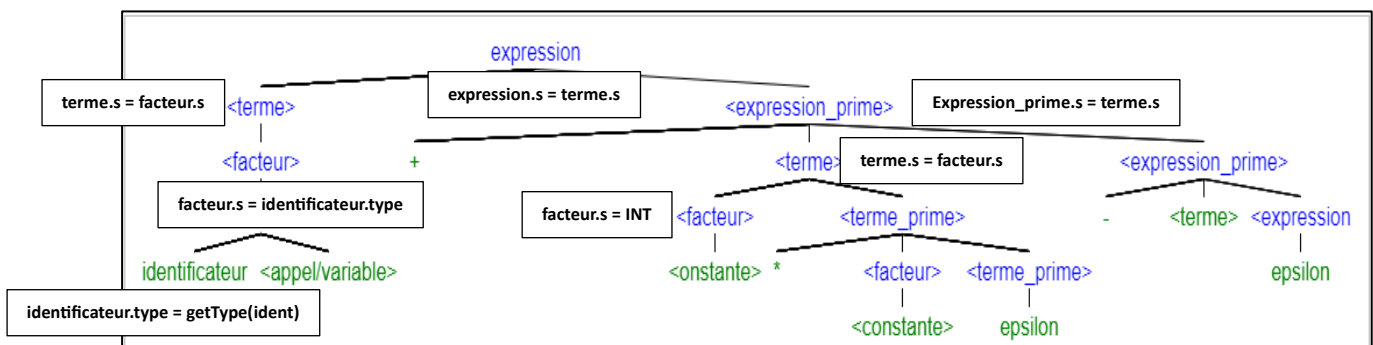
<expression> : <terme> {expression.s = terme.s} <expression_prime> | -<expression> {expression.s = expression.l.s}

<expression_prime> : + <terme> {expression_prime.s = terme.s} <expression_prime> | - <terme> {expression_prime.s = terme.s} <expression_prime> | epsilon

<terme> : <facteur> {terme.s = facteur.s} <terme_prime>

<terme_prime> : * <facteur> {terme_prime.s = facteur.s} <terme_prime> | / <facteur> {terme_prime.s = facteur.s} <terme_prime> | epsilon

<facteur> : **identificateur** {facteur.s = identificateur.type} <appel/variable> | <constante> {facteur.s = INT} | <char> {facteur.s = CAR} | (<expression>) {facteur.s = expression.s}



5. Les tests

- Utilisation d'une variable non déclarée

```

1  int t[3], i;
2  car q;
3
4  int w(car b, int a){
5      car s[2];
6      int n[1];
7  }
8
9  void test(int x, car y){
10     int m[2];
11     car e;
12
13     m[1] = b;
14 }

```

Une variable qui n'est pas déclarée n'est ni comme globale ni comme variable locale ni comme paramètres est non déclarée.

On voit bien l'erreur dans la ligne 13.

```

liste_instructions
| instruction
| | appel_affectation
| | | expression
| | | | terme
| | | | | facteur
| | | | | terme_prime
| | | | expression_prime
| | | expression
| | | | terme
| | | | | facteur
| | | | | 13) variable non definie b
| | | | | appel_variable
| | | | | terme_prime
| | | | expression_prime
| | liste_instructions
| liste_fonctions
Votre code est correct synthaxiquement

```

- Redéfinition d'une variable

```

1  int t[3], i;
2  car q;
3  int q[5];
4
5  int w(car b, int b, int a){
6      car s[2];
7      int n[1];
8
9      int s;
10 }

```

La redéfinition d'une variable globale localement est allouée.

```

programme prime
| declarateur_prime
| 3) redifinition de q
| liste_declarateurs_prim

```

```

liste_params
|param
|liste_params_prime
|param
---|5) redefinition de b
|liste_params_prime
|param
|liste_params_prime

```

```

|declarateur
|declarateur_prime
|liste_declarateurs_prim
liste_declarations
|declaration
|liste_declarateurs
|declarateur
---|9) redifinition de s
|declarateur_prime
|liste_declarateurs_prim
|liste_declarations
liste_instructions
liste_fonctions
Votre code est correct syntaxiquement

```

- Affectation des variables à type non compatibles

```

1 int t[3], i;
2 car q;
3
4 int w(car a, int b){
5     car s[2];
6     int n[1];
7
8     s[1] = 4;
9
10    a = b;
11
12    n[0] = '\n';
13 }

```

Dans ce cas même si dans le langage C on accepte l'affectation d'un entier a un caractère et vice versa (par la conversion implicite en code ascii). Nous choisissons de ne pas allouer cette propriété.

```

|terme_prime
|expression_prime
---|8) type non compatible
|liste_instructions
|instruction

```

```

|appel_variable
|terme_prime
|expression_prime
---|10) type non compatible
|liste_instructions

```

```

|terme_prime
|expression_prime
---|12) type non compatible
|liste_instructions
|liste_fonctions
Votre code est correct syntaxiquement

```

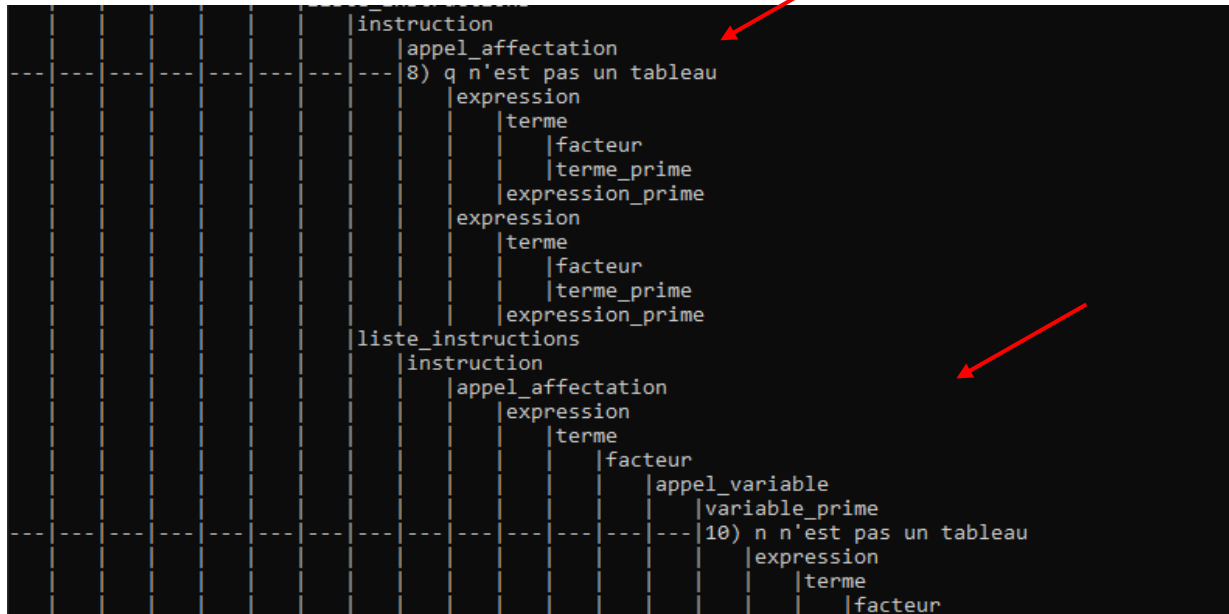
- Appel à une variable comme tableau

```

1  int t[3], i;
2  car q;
3
4  int w(car a, int b){
5      car s[2];
6      int n;
7
8      q[1] = 'c';
9
10     n = n[1];
11 }

```

Ici nous avons déclaré q comme variable globale et n comme locales les deux ne sont pas des tableaux.



- Appel à une variable comme fonction

```

1  int t[3], i;
2  car q;
3
4  int w(car a, int b){
5      car s[2];
6      int n;
7
8      s[1] = q();
9
10     i = n();
11 }

```

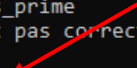
Ici q et s ne sont pas des tableaux.

- Appel d'une fonction avec un nombre de paramètres incorrect

```

1  int t[3], i;
2  car q;
3
4  int somme(int a, int b){
5      return a+b;
6  }
7
8  void fun(int a){
9  }
10
11 int w(car a, int b){
12     car s[2];
13     int n;
14
15     n = somme( 3, 4, 3);
16
17     q = fun(2);
18 }

```



- Appel à une fonction avec types de paramètres non corrects



```
C:\Users\hp\Desktop\Analyse semantique\code.txt
3 int somme(int a, int b){
4     return a+b;
5 }
6
7 int w(car a, int b){
8
9     i = somme(1, 'A');
10 }
```

Dans ce cas on fait appel au fonction somme avec des paramètres de types incorrects.

```

expression_prime
liste_expressions_prime
(9) types des parametres ne sont pas corrects
terme_prime
expression_prime
liste_instructions
liste_fonctions
Votre code est correct syntaxiquement

```

- Appel d'une fonction comme variable

```

1 int i;
2
3 int f(car a, int b){
4
5 }
6
7 void test(){
8     f = 3;
9 }
10

```

```

instruction
appel_affectation
(8) fest une fonction.
expression
terme
facteur
terme_prime
expression_prime
liste_instructions
liste_fonctions
Votre code est correct syntaxiquement

```

- Vérification type de retour d'une fonction

```

1 int i;
2
3 int f(car a, int b){
4     return 's';
5 }

```



```
expression
|terme
|facteur
|terme_prime
|expression_prime
---|---|---|---|---|---|4) type de retour non compatible
|liste_instructions
|liste_fonctions
Votre code est correct syntaxiquement
```

Conclusion

Ce projet de réalisation d'un compilateur C++ nous a permis d'explorer les différentes phases de la compilation, à savoir l'analyse lexicale, syntaxique et sémantique.

Durant ce projet, nous avons implémenté :

- Un analyseur lexical capable de découper le code source en unités lexicales (jetons) et de les identifier.
- Un analyseur syntaxique qui vérifie la structure du code source et construit un arbre de syntaxe abstrait.
- Un analyseur sémantique qui effectue des vérifications de type, de déclaration et de cohérence du code source.

Notre compilateur C++ est capable de compiler des programmes simples et nous a permis de les valider. Il nous a également permis de développer nos compétences en programmation, en résolution de problèmes et en gestion de projet.

Cependant, nous sommes conscients des limitations de notre implémentation actuelle. Le compilateur ne prend pas en charge toutes les fonctionnalités du langage C++ et ne gère pas certains cas complexes. Nous sommes motivés à améliorer notre compilateur dans les versions futures en ajoutant de nouvelles fonctionnalités, en optimisant le code généré et en améliorant la gestion des erreurs.

Ce projet a été une expérience enrichissante qui nous a permis de comprendre les aspects fondamentaux de la compilation et d'apprécier la complexité du développement d'un compilateur.

Nous remercions tous ceux qui nous ont aidés et soutenus dans la réalisation de ce projet.