**Rapport de Projet : Compilateur Minimal pour la Structure do-while en Langage C**

Nombre de groupe :

* **AZIZI MOHAMMED ABDELAZIZ**
* **YAHIA MAMOUNE Ismail Abdelrahim**
* **MEZOUAR NADIR**

**1. Introduction**

L'objectif de ce projet est la conception et la réalisation d'un compilateur partiel permettant de reconnaître et de vérifier la syntaxe d'une structure de contrôle do-while en langage C. Ce compilateur effectue deux étapes majeures : une **analyse lexicale**, puis une **analyse syntaxique**.

**2. Analyse Lexicale**

**2.1 Objectif**

L'analyseur lexical a pour rôle de lire le code source caractère par caractère, de regrouper les séquences valides en **unités lexicales** (U.L.) et de les associer à un **code symbolique**.

**2.2 Fonctionnement**

* Lecture du fichier source
* Ignorer les espaces, tabulations et commentaires (//, /\* \*/)
* Regrouper les caractères valides en lexèmes (identifiants, mots-clés, opérateurs, etc.)
* Attribution d’un code symbolique pour chaque lexème
* Enregistrement dans une liste chaînée des unités lexicales

**2.3 – Reconnaissance des Unités Lexicales**

La reconnaissance des unités lexicales est réalisée à l’aide d’un **analyseur lexical** implémenté dans le fichier analyseur\_lexical.c. Cet analyseur parcourt le code source caractère par caractère pour identifier et extraire les éléments du langage source selon les règles lexicales définies. Chaque lexème reconnu est transformé en **unité lexicale (UL)** contenant :

* le **lexème** (ex. : do, i, ++, ;, 5)
* le **code** correspondant (défini dans analyseur\_lexical.h)
* le **numéro de ligne**

**Méthode de reconnaissance utilisée :**

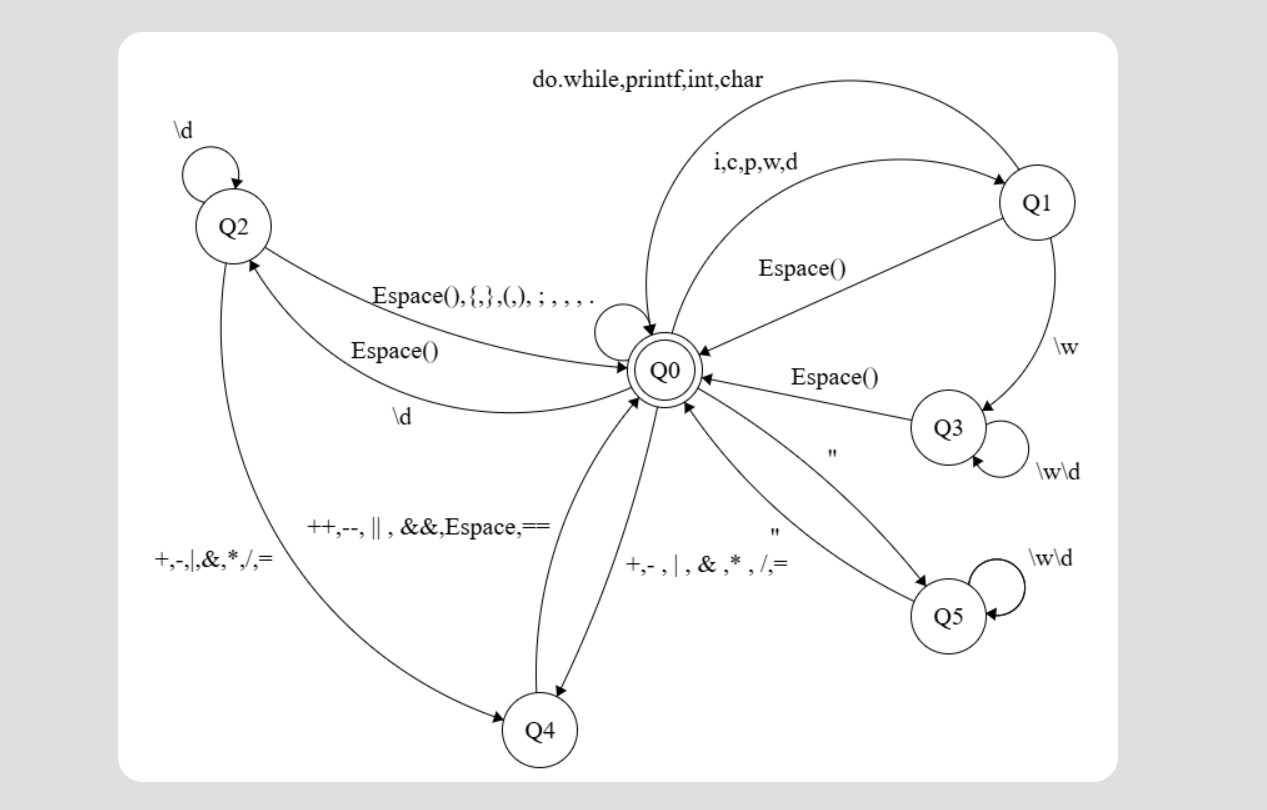
1. **Itération caractère par caractère** dans la chaîne lue depuis le fichier source (read\_file).
2. **Classification selon la nature du caractère courant** :
   * Lettre → début d’un mot-clé ou identifiant.
   * Chiffre → constante entière.
   * Symbole → opérateur, délimiteur, parenthèse, etc.
   * Slash suivi de slash (//) ou étoile (/\*) → début d’un **commentaire**.
3. **Utilisation de fonctions auxiliaires** comme isalpha, isdigit, isspace, pour déterminer le type du caractère.
4. **Buffer temporaire** pour accumuler les caractères d’un lexème (ex. : nom de variable, mot-clé).
5. **Comparaison avec les mots-clés réservés** (do, while, int, char, etc.) pour leur attribuer des codes spécifiques.
6. **Ajout de l’unité lexicale** à la liste via la fonction inserer\_fin(...).
7. **Gestion des erreurs** : tout caractère non reconnu déclenche la génération d’un message d’erreur enregistré dans une liste (SU).

**Exemples de reconnaissance :**

* Chaîne "int" → Reconnu comme mot-clé avec le code CODE\_INT = 103.
* Symbole ++ → Reconnu comme opérateur d’incrémentation avec le code CODE\_PLUSPLUS = 115.
* Identifiant i → Reconnu comme CODE\_ID = 102.
* Constante 5 → Reconnu comme CODE\_INT\_CONST = 125.

L’analyseur est également capable d’ignorer les **commentaires** (ligne et bloc) tout en tenant compte des retours à la ligne pour un traçage correct des erreurs.

**2.4 – Automate etat finie : analyseur lexical :**



\d signifie caractère numérique quelconque

\w signifie caractère d’une lettrequelconque

### 2.5– Attribution des codes aux U.L. (Unités Lexicales) :

Dans le cadre de ce projet, chaque unité lexicale (mot-clé, identifiant, constante, opérateur, délimiteur, etc.) rencontrée dans le code source est reconnue et codée à l’aide d’un entier unique. Ce code est utilisé pour simplifier l’analyse syntaxique et identifier rapidement le type de lexème.

L’attribution des codes est faite manuellement à travers des #define dans le fichier analyseur\_lexical.h. Chaque code est unique et correspond à un symbole lexical bien déterminé.

| **Unité Lexicale (Nom symbolique)** | **Exemple de Lexème** | **Code (Type\_UL)** |
| --- | --- | --- |
| CODE\_DO | do | 100 |
| CODE\_WHILE | while | 101 |
| CODE\_ID | i, index, somme | 102 |
| CODE\_INT | int | 103 |
| CODE\_CHAR | char | 104 |
| CODE\_PV | ; | 105 |
| CODE\_ACCOLADE\_OUVRANTE | { | 106 |
| CODE\_ACCOLADE\_FERMANTE | } | 107 |
| CODE\_PARENTHESE\_OUVRANTE | ( | 108 |
| CODE\_PARENTHESE\_FERMANTE | ) | 109 |
| CODE\_AFFECTATION | = | 110 |
| CODE\_PLUS | + | 111 |
| CODE\_MINUS | - | 112 |
| CODE\_MULTIPLY | \* | 113 |
| CODE\_DIVIDE | / | 114 |
| CODE\_PLUSPLUS | ++ | 115 |
| CODE\_MINUSMINUS | -- | 116 |
| CODE\_EQ | == | 117 |
| CODE\_NEQ | != | 118 |
| CODE\_INF | < | 119 |
| CODE\_SUP | > | 120 |
| CODE\_LEQ | <= | 121 |
| CODE\_GEQ | >= | 122 |
| CODE\_AND | && | 123 |
| CODE\_OR | ` |  |
| CODE\_INT\_CONST | 5, 10, 999 | 125 |
| CODE\_STRING | "Bonjour" | 126 |
| CODE\_VIRGULE | , | 127 |
| CODE\_POINT | . | 128 |
| CODE\_EPSILON (interne) | — | 129 |
| FIN\_SUITE\_UL | # (fin de fichier) | 999 |
| ERREUR | %, @, etc. | -1 |

### 2.5 – Structure d’une Unité Lexicale (U.L.) :

### Dans le programme, une ****unité lexicale**** (UL) est représentée par une ****structure en langage C**** définie dans le fichier lexime.h :

### typedef struct Unite\_Lexicale {

### char Lexeme[LEXEME\_SIZE];

### int Code;

### int Ligne;

### struct Unite\_Lexicale\* Suivant;

### } UL;

### 2.6 – ****Algorithme simplifié de l’analyseur lexical :****

Voici les étapes principales :

1. Lire le **texte source** caractère par caractère.
2. Ignorer les **espaces**, **sauts de ligne**, et **commentaires** (//, /\* \*/).
3. Reconnaitre :
   * les **mots-clés** (ex : do, while, int , char..),
   * les **identifiants** (noms de variables),
   * les **opérateurs** (++, =, !=, >= , <= , >, <.),
   * les **délimiteurs** (;, {, }.(,)., .),
   * les **littéraux** (5, "texte"...).
4. Créer pour chaque élément une **UL** avec lexème, code et ligne.
5. Signaler toute erreur lexicale (caractère non reconnu).

**Exemple analysé :**

Code source fourni (code\_source.txt) :

Analyse lexicale avec succès :

do {

int a = 5;

char nom = "compilateur do\_whileen c"

++a;

--a;

} while (i < 3); // Tant que i est inférieur à 3

=== Analyseur Lexical ===

Lexeme: [do] Code: [100] Ligne: [1]

Lexeme: [{] Code: [106] Ligne: [1]

Lexeme: [int] Code: [103] Ligne: [3]

Lexeme: [a] Code: [102] Ligne: [3]

Lexeme: [=] Code: [110] Ligne: [3]

Lexeme: [5] Code: [125] Ligne: [3]

Lexeme: [;] Code: [105] Ligne: [3]

Lexeme: [char] Code: [104] Ligne: [4]

Lexeme: [nom] Code: [102] Ligne: [4]

Lexeme: [=] Code: [110] Ligne: [4]

Lexeme: ["compilateur do\_while C"] Code: [126] Ligne: [4]

Lexeme: [++] Code: [115] Ligne: [5]

Lexeme: [a] Code: [102] Ligne: [5]

Lexeme: [;] Code: [105] Ligne: [5]

Lexeme: [--] Code: [116] Ligne: [6]

Lexeme: [a] Code: [102] Ligne: [6]

Lexeme: [;] Code: [105] Ligne: [6]

Lexeme: [}] Code: [107] Ligne: [7]

Lexeme: [while] Code: [101] Ligne: [7]

Lexeme: [(] Code: [108] Ligne: [7]

Lexeme: [i] Code: [102] Ligne: [7]

Lexeme: [<] Code: [119] Ligne: [7]

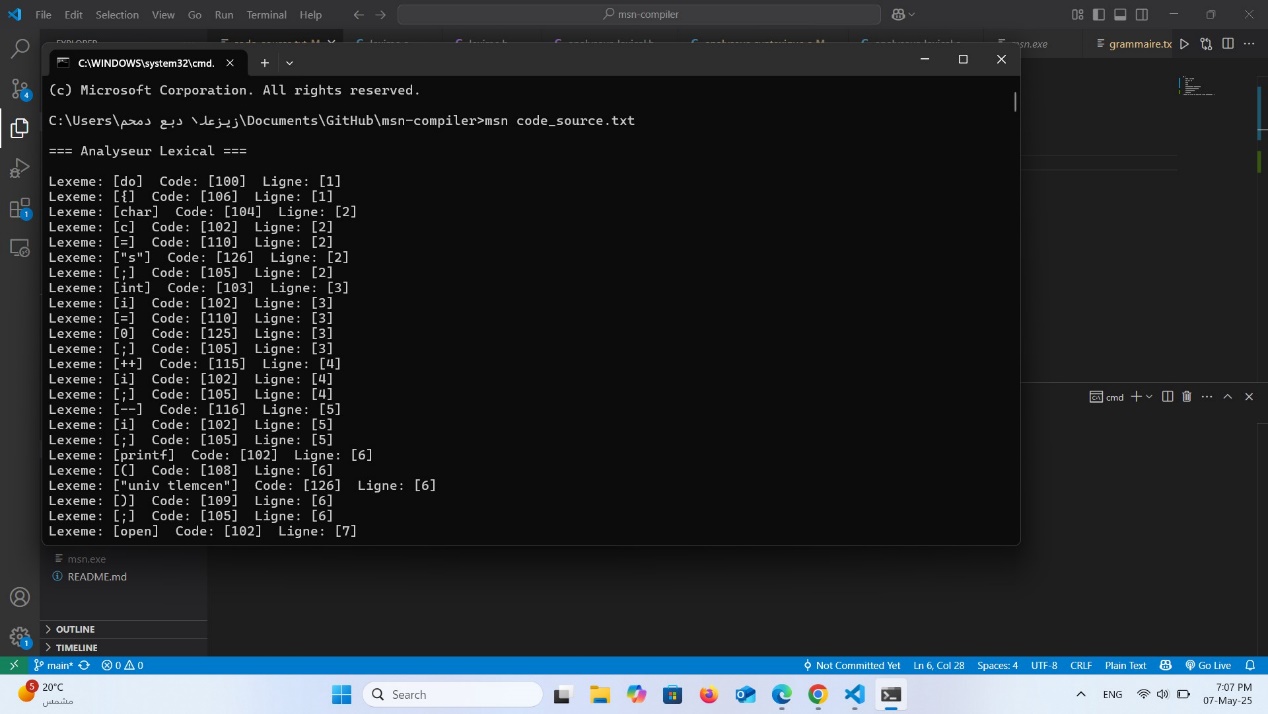
Lexeme: [3] Code: [125] Ligne: [7]

Lexeme: [)] Code: [109] Ligne: [7]

Lexeme: [;] Code: [105] Ligne: [7]

Lexeme: [#] Code: [999] Ligne: [7]

Aucune erreur lexicale.



compilateur peut reconnaitre ce code

do {  
   char c = "s";  
   int i = 0;  
   ++i;  
   --i;  
   printf("univ tlemcen");  
   open("fichier.txt");  
   avg(1,2,3,70);  
   clear();  
} while (i < 3); // Tant que i est inférieur à 3

**3. Analyse Syntaxique**

**3.1 Objectif**

L’objectif est de construire une **table d’analyse syntaxique prédictive** pour le langage restreint do { ... } while ( ... );, dans lequel on peut avoir des déclarations de variables, des affectations, des expressions conditionnelles, des incrémentations ou décrémentations, etc.

Le langage est défini par une **grammaire LL(1)** simplifiée avec :

* un **axiome** : S → do { statements } while ( condition );
* des **non-terminaux** pour les instructions, expressions, conditions, etc.

### 3.2 – Liste des non-terminaux :

### a travers des #define dans le fichier analyseur\_syntaxique.h

| **Code** | **Nom du Non-Terminal** | **Symbole associé** |
| --- | --- | --- |
| 200 | NT\_DO\_WHILE | <do\_while> |
| 201 | NT\_STATEMENTS | <statements> |
| 202 | NT\_STATEMENT | <statement> |
| 203 | NT\_DECLARE\_STMT | <declare\_stmt> |
| 204 | NT\_TYPE | <type> |
| 205 | NT\_EXPRESSION | <expression> |
| 206 | NT\_OPERAND | <operand> |
| 207 | NT\_OPERATION | <operation> |
| 208 | NT\_INCREMENT\_STMT | <increment\_stmt> |
| 209 | NT\_DECREMENT\_STMT | <decrement\_stmt> |
| 210 | NT\_CONDITION | <condition> |
| 211 | NT\_REL\_OP | <rel\_op> |
| 212 | NT\_BOOL\_OP | <bool\_op> |
| 213 | NT\_APPELLE\_FONCTION | <appelle\_fonction> |
| 214 | NT\_ARGS | <args> |
| 215 | NT\_SEPARATEUR | <separateur> |

### 3.3 Table des règles de production (grammaire) :

### a travers le fichier analyseur\_syntaxique.c

| **N° RP** | **Règle de production** |
| --- | --- |
| 0 | <do\_while> → do { <statements> } while ( <condition> ) ; |
| 1 | <statements> → <statement> <statements> |
| 2 | <statements> → ε |
| 3 | <statement> → <increment\_stmt> |
| 4 | <statement> → <decrement\_stmt> |
| 5 | <statement> → <declare\_stmt> |
| 6 | <declare\_stmt> → int id = int\_const ; |
| 7 | <declare\_stmt> → char id = string ; |
| 8 | <expression> → <type> id ; |
| 9 | <type> → int |
| 10 | <type> → char |
| 11 | <condition> → <operand> <operation> <operand> |
| 12 | <operation> → + |
| 13 | <operation> → - |
| 14 | <operation> → \* |
| 15 | <operation> → / |
| 16 | <increment\_stmt> → ++ id; |
| 17 | <decrement\_stmt> → -- id ; |
| 18 | <condition> → <operand> <rel\_op> <operand> |
| 19 | <rel\_op> → == |
| 20 | <rel\_op> → != |
| 21 | <rel\_op> → < |
| 22 | <rel\_op> → > |
| 23 | <rel\_op> → <= |
| 24 | <rel\_op> → >= |
| 25 | <bool\_op> → && |
| 26 | `<bool\_op> → |
| 27 | <operand> → id |
| 28 | <operand> → int\_const |
| 29 | <increment\_stmt> → ++ id ; |
| 30 | <decrement\_stmt> → -- id ; |
| 31 | <function\_call> → id ( <args> ) ; |
| 32 | <args> → ε |
| 33 | <args> → string <separateur> <args> |
| 34 | <args> → <operand> <separateur> <args> |
| 35 | <separateur> → , |
| 36 | <separateur> → ε |

### code source:

static const int RPs[][10] = {

    /\*  0 \*/ { CODE\_DO, CODE\_ACCOLADE\_OUVRANTE, NT\_STATEMENTS, CODE\_ACCOLADE\_FERMANTE,

              CODE\_WHILE, CODE\_PARENTHESE\_OUVRANTE, NT\_CONDITION, CODE\_PARENTHESE\_FERMANTE, CODE\_PV, -1 },

    /\*  1 \*/ { NT\_STATEMENT, NT\_STATEMENTS, -1 },

    /\*  2 \*/ { CODE\_EPSILON, -1 },  // ε production for <statements>

    /\*  3 \*/ { NT\_INCREMENT\_STMT, -1 },

    /\*  4 \*/ { NT\_DECREMENT\_STMT, -1 },

    /\*  5 \*/ { NT\_DECLARE\_STMT, -1 },

    /\*  6 \*/ { CODE\_INT, CODE\_ID, CODE\_AFFECTATION, CODE\_INT\_CONST, CODE\_PV, -1 },

    /\*  7 \*/ { CODE\_CHAR, CODE\_ID, CODE\_AFFECTATION, CODE\_STRING, CODE\_PV, -1 },

    /\*  8 \*/ { NT\_TYPE, CODE\_ID, CODE\_PV, -1 },

    /\*  9 \*/ { CODE\_INT, -1 },

    /\* 10 \*/ { CODE\_CHAR, -1 },

    /\* 11 \*/ { NT\_OPERAND, NT\_OPERATION, NT\_OPERAND, -1 },

    /\* 12 \*/ { CODE\_PLUS, -1 },

    /\* 13 \*/ { CODE\_MINUS, -1 },

    /\* 14 \*/ { CODE\_MULTIPLY, -1 },

    /\* 15 \*/ { CODE\_DIVIDE, -1 },

    /\* 16 \*/ { CODE\_ID, CODE\_PLUSPLUS, CODE\_PV, -1 },

    /\* 17 \*/ { CODE\_ID, CODE\_MINUSMINUS, CODE\_PV, -1 },

    /\* 18 \*/ { NT\_OPERAND, NT\_REL\_OP, NT\_OPERAND, -1 },

    /\* 19 \*/ { CODE\_EQ, -1 },

    /\* 20 \*/ { CODE\_NEQ, -1 },

    /\* 21 \*/ { CODE\_INF, -1 },

    /\* 22 \*/ { CODE\_SUP, -1 },

    /\* 23 \*/ { CODE\_LEQ, -1 },

    /\* 24 \*/ { CODE\_GEQ, -1 },

    /\* 25 \*/ { CODE\_AND, -1 },

    /\* 26 \*/ { CODE\_OR, -1 },

    // NT\_OPERAND

    /\* 27 \*/ {CODE\_ID, -1 },

    /\* 28 \*/ {CODE\_INT\_CONST, -1 },

    // NT\_INC

    /\* 29 \*/ {CODE\_PLUSPLUS,CODE\_ID ,CODE\_PV, -1},

    // NT\_DEC

    /\* 30 \*/ {CODE\_MINUSMINUS,CODE\_ID ,CODE\_PV, -1},

    // NT\_APPEL FONCTION

    /\*31 \*/ {CODE\_ID,CODE\_PARENTHESE\_OUVRANTE,NT\_ARGS,CODE\_PARENTHESE\_FERMANTE,CODE\_PV,-1},

    // NT\_ARGS

    /\*32\*/  {CODE\_EPSILON,-1},

    /\*33\*/ {CODE\_STRING,NT\_SEPARATEUR,NT\_ARGS,-1},

    /\*34\*/ {NT\_OPERAND,NT\_SEPARATEUR,NT\_ARGS,-1},

    // NT\_SEPARATEUR

    /\*35\*/ {CODE\_VIRGULE,-1},

    /\*36\*/ {CODE\_EPSILON,-1}  };

### 3.4 Calcul des ensembles FIRST et FOLLOW :

#### 🔹 ****Ensembles FIRST des Non Terminaux****

| **Non terminal** | **FIRST** |
| --- | --- |
| <do\_while> | { do } |
| <statements> | { int, char, id, ++, --, ε } |
| <statement> | { int, char, id, ++, -- } |
| <declare\_stmt> | { int, char } |
| <type> | { int, char } |
| <expression> | { int, char } |
| <operand> | { id, int\_const } |
| <operation> | { +, -, \*, / } |
| <increment\_stmt> | { id, ++ } |
| <decrement\_stmt> | { id, -- } |
| <condition> | { id, int\_const } |
| <rel\_op> | { ==, !=, <, >, <=, >= } |
| <bool\_op> | { &&, ` |
| <function\_call> | { id } |
| <args> | { string, id, int\_const, ε } |
| <separateur> | { ,, ε } |

#### 🔹 ****Ensembles FOLLOW des Non Terminaux****

| **Non terminal** | **FOLLOW** |
| --- | --- |
| <do\_while> | { # } |
| <statements> | { } } |
| <statement> | { int, char, id, ++, --, } } |
| <declare\_stmt> | { int, char, id, ++, --, } } |
| <type> | { id } |
| <expression> | { ; } |
| <operand> | { ==, !=, <, >, <=, >=, ;, , } |
| <operation> | { id, int\_const } |
| <increment\_stmt> | { int, char, id, ++, --, } } |
| <decrement\_stmt> | { int, char, id, ++, --, } } |
| <condition> | { ) } |
| <rel\_op> | { id, int\_const } |
| <bool\_op> | { id, int\_const } |
| <function\_call> | { ; } |
| <args> | { ) } |
| <separateur> | { id, int\_const, string, ) } |

### 3.5 Table d'analyse :

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Non Terminal \ Terminal | do | while | id | int | char | ; | { | } | ( | ) | = | + | - | \* | / | ++ | -- | == | != | < | > | <= | >= | && | || | int\_const | string | , | . |
| <do\_while> | 0 | er | er | er | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | Er |
| <statements> | er | er | 1 | 1 | 1 | Er | 2 | 2 | er | er | er | er | er | er | er | 1 | 1 | er | er | er | er | er | er | er | er | er | er | er | er |
| <statement> | er | er | 31 | 5 | 5 | Er | er | er | er | er | er | er | er | er | er | 3 | 4 | er | er | 3 | 4 | er | er | er | er | er | er | er | er |
| <declare\_stmt> | er | er | er | 6 | 7 | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er |
| <type> | er | er | 7 | er | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er |
| <expression> | 8 | er | 9 | er | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er |
| <operand> | er | er | 27 | 12 | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | 28 | er | er | er | er |
| <operation> | er | er | er | er | er | Er | er | er | er | er | er | er | er | er | er | er | er | 19 | 20 | 21 | 22 | 23 | 24 | er | er | er | er | er | er |
| <increment\_stmt> | 17 | er | er | er | er | Er | er | er | er | er | er | er | er | er | er | 29 | 30 | er | er | er | er | er | er | er | er | er | er | er | er |
| <decrement\_stmt> | 18 | er | er | er | er | Er | er | er | er | er | er | er | er | er | er | er | 30 | er | er | er | er | er | er | er | er | er | er | er | er |
| <condition> | er | er | 11 | er | er | Er | er | er | 19 | 20 | 21 | 22 | er | er | er | er | er | er | er | er | er | 11 | er | er | er | er | er | er | er |
| <rel\_op> | er | er | 25 | 26 | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er |
| <bool\_op> | er | er | er | er | er | Er | er | er | er | er | er | er | er | 27 | 28 | er | er | er | er | er | er | er | er | er | er | er | er | er | er |
| <appel fonction> | er | er | 31 | er | er | Er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | 33 | er | er | er |
| <args> | er | er | 34 | er | er | Er | er | er | 32 | 32 | 32 | er | er | er | er | er | er | er | er | er | er | er | er | er | 34 | 33 | er | er | er |
| <separateur> | er | er | er | er | er | Er | er | er | er | 36 | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | er | 35 | er |  |

### 3.6 PROGRAMMATION DE L’ANALYSEUR SYNTAXIQUE (LL1) :

### L'objectif est d’implémenter un analyseur syntaxique descendant non récursif basé sur une analyse LL(1) pour valider la structure syntaxique d’une boucle do...while, en s’appuyant sur la table d’analyse construite dans la partie précédente.

### Structures de données utilisées

### UL (Unité Lexicale) : définie dans lexime.h pour stocker un lexème, son code et son numéro de ligne.

### ULPile : une structure pile implémentée via une liste chaînée pour gérer les symboles de la pile d’analyse.

### Table d’analyse : tableau int Table\_DAnalyse[NB\_NON\_TERMINAUX][NB\_TERMINAUX], contenant les indices des règles de production.

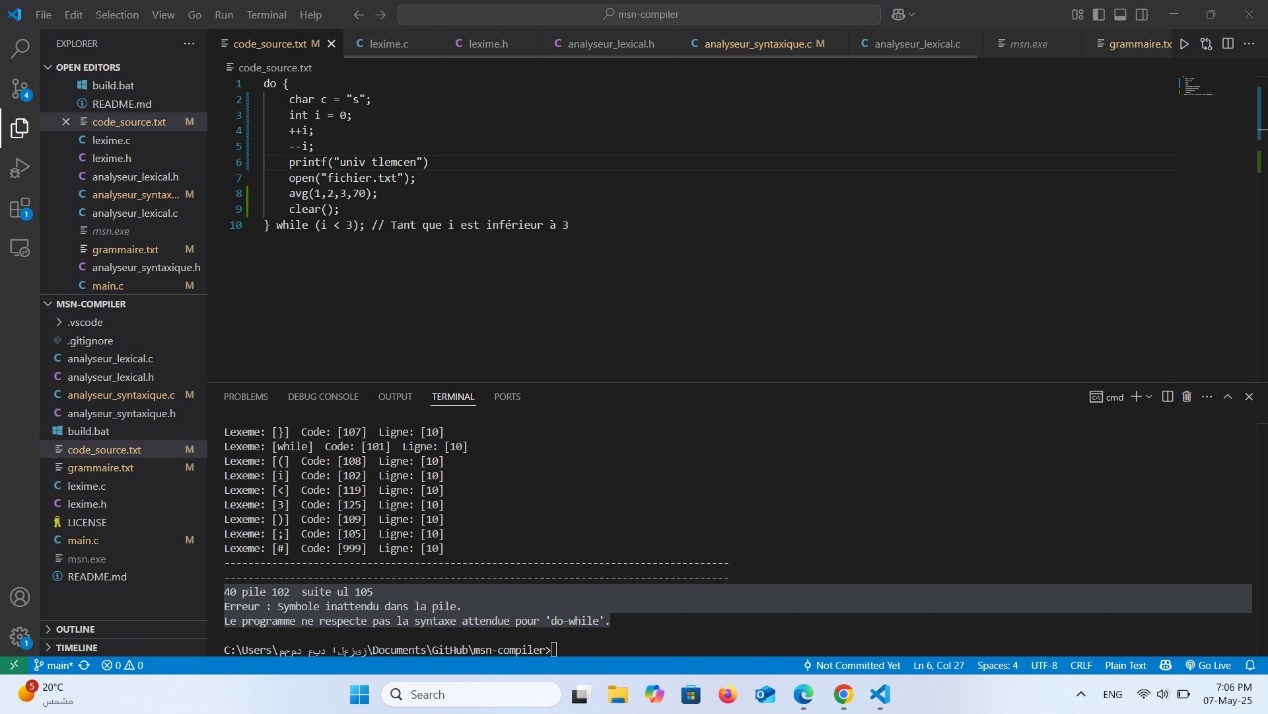
### RPs : matrice contenant les règles de production sous forme de suites d’entiers (symboles).

### Exemple de déroulement d’exécution :

### Code source à analyser :

do {  
    char c = "s";  
    int i = 0;  
    ++i;  
    --i;  
    printf("univ tlemcen")  
    open("fichier.txt");  
    avg(1,2,3,70);  
    clear();  
} while (i < 3); // Tant que i est inférieur à 3

=== Analyseur Syntaxique ===

Analyse syntaxique avec erreur

Analyseur Syntaxique Structure do-while correcte

do {  
    char c = "s";  
    int i = 0;  
    ++i;  
    --i;  
    printf("univ tlemcen")  
    open("fichier.txt");  
    avg(1,2,3,70);  
    clear();  
} while (i < 3); // Tant que i est inférieur à 3

