

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 :

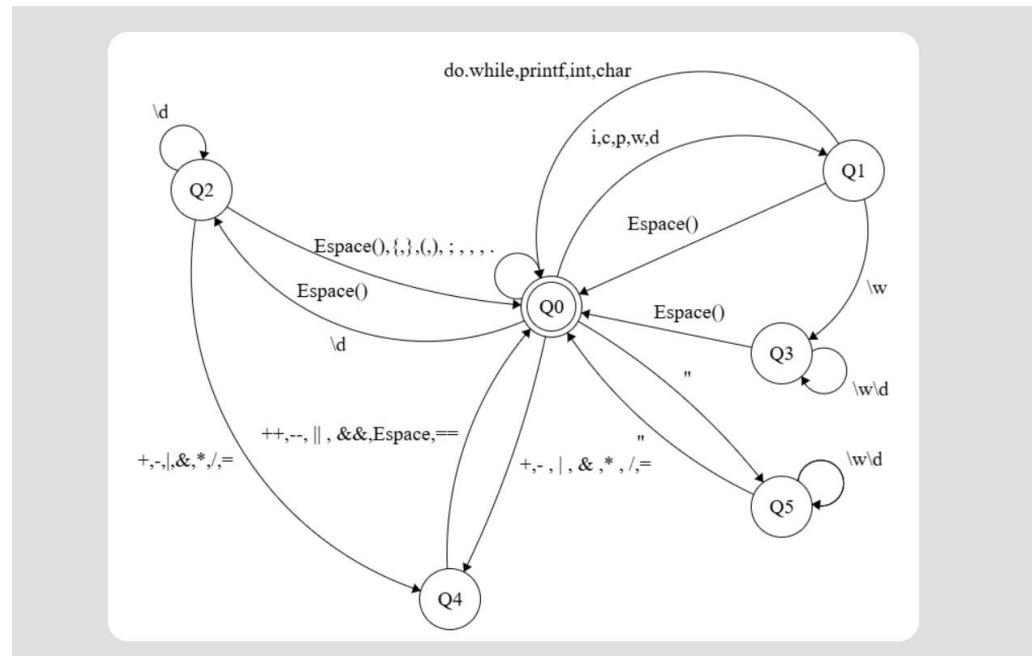
1. **Itération caractère par caractère** dans la chaîne lue depuis le fichier source (`read_file`).
2. **Classification selon la nature du caractère courant :**
 - o Lettre → début d'un mot-clé ou identifiant.
 - o Chiffre → constante entière.
 - o Symbole → opérateur, délimiteur, parenthèse, etc.
 - o 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 état finie : analyseur lexical :



\d signifie caractère numérique quelconque

\w signifie caractère d'une lettre quelconque

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

Unité Lexicale (Nom symbolique)	Exemple de Lexème	Code (Type_UL)
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. Reconnaître :
 - o les **mots-clés** (ex : do, while, int , char..),
 - o les **identifiants** (noms de variables),
 - o les **opérateurs** (++, !=, >= , <= , >, <.),
 - o les **délimiteurs** (;, {, }.(), .),
 - o 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.

```

File Edit Selection View Go Run Terminal Help ↵ msن Compiler
C:\WINDOWS\system32\cmd. x + v
(c) Microsoft Corporation. All rights reserved.
C:\Users\msn\Documents\GitHub\msn-compiler>msn code_source.txt
== Analyseur Lexical ==
Lexeme: [do] Code: [100] Ligne: [1]
Lexeme: [i] Code: [106] Ligne: [1]
Lexeme: [char] Code: [100] Ligne: [2]
Lexeme: [c] Code: [102] Ligne: [2]
Lexeme: [z] Code: [110] Ligne: [2]
Lexeme: ["s"] Code: [126] Ligne: [2]
Lexeme: [:] Code: [105] Ligne: [2]
Lexeme: [int] Code: [103] Ligne: [3]
Lexeme: [i] Code: [102] Ligne: [3]
Lexeme: [=] Code: [110] Ligne: [3]
Lexeme: [0] Code: [125] Ligne: [3]
Lexeme: [,] Code: [105] Ligne: [3]
Lexeme: [+] Code: [115] Ligne: [4]
Lexeme: [i] Code: [102] Ligne: [4]
Lexeme: [:] Code: [105] Ligne: [4]
Lexeme: [-] Code: [116] Ligne: [5]
Lexeme: [i] Code: [102] Ligne: [5]
Lexeme: [,] Code: [105] Ligne: [5]
Lexeme: [printf] Code: [102] Ligne: [6]
Lexeme: [c] Code: [108] Ligne: [6]
Lexeme: [int] Code: [105] Ligne: [6]
Lexeme: [)] Code: [109] Ligne: [6]
Lexeme: [,] Code: [105] Ligne: [6]
Lexeme: [open] Code: [102] Ligne: [7]

msn.exe
 README.md
 OUTLINE
 TIMELINE
 20°C

```

compilateur peut reconnaître 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	$\langle \text{do_while} \rangle \rightarrow \text{do } \{ \langle \text{statements} \rangle \} \text{ while } (\langle \text{condition} \rangle) ;$
1	$\langle \text{statements} \rangle \rightarrow \langle \text{statement} \rangle \langle \text{statements} \rangle$
2	$\langle \text{statements} \rangle \rightarrow \epsilon$
3	$\langle \text{statement} \rangle \rightarrow \langle \text{increment_stmt} \rangle$
4	$\langle \text{statement} \rangle \rightarrow \langle \text{decrement_stmt} \rangle$
5	$\langle \text{statement} \rangle \rightarrow \langle \text{declare_stmt} \rangle$
6	$\langle \text{declare_stmt} \rangle \rightarrow \text{int id} = \text{int_const} ;$
7	$\langle \text{declare_stmt} \rangle \rightarrow \text{char id} = \text{string} ;$
8	$\langle \text{expression} \rangle \rightarrow \langle \text{type} \rangle \text{id} ;$
9	$\langle \text{type} \rangle \rightarrow \text{int}$
10	$\langle \text{type} \rangle \rightarrow \text{char}$
11	$\langle \text{condition} \rangle \rightarrow \langle \text{operand} \rangle \langle \text{operation} \rangle \langle \text{operand} \rangle$
12	$\langle \text{operation} \rangle \rightarrow +$
13	$\langle \text{operation} \rangle \rightarrow -$
14	$\langle \text{operation} \rangle \rightarrow *$
15	$\langle \text{operation} \rangle \rightarrow /$
16	$\langle \text{increment_stmt} \rangle \rightarrow ++ \text{id} ;$
17	$\langle \text{decrement_stmt} \rangle \rightarrow -- \text{id} ;$
18	$\langle \text{condition} \rangle \rightarrow \langle \text{operand} \rangle \langle \text{rel_op} \rangle \langle \text{operand} \rangle$
19	$\langle \text{rel_op} \rangle \rightarrow ==$
20	$\langle \text{rel_op} \rangle \rightarrow !=$
21	$\langle \text{rel_op} \rangle \rightarrow <$
22	$\langle \text{rel_op} \rangle \rightarrow >$
23	$\langle \text{rel_op} \rangle \rightarrow <=$
24	$\langle \text{rel_op} \rangle \rightarrow >=$
25	$\langle \text{bool_op} \rangle \rightarrow \&&$
26	$\langle \text{bool_op} \rangle \rightarrow \neg$
27	$\langle \text{operand} \rangle \rightarrow \text{id}$
28	$\langle \text{operand} \rangle \rightarrow \text{int_const}$
29	$\langle \text{increment_stmt} \rangle \rightarrow ++ \text{id} ;$
30	$\langle \text{decrement_stmt} \rangle \rightarrow -- \text{id} ;$
31	$\langle \text{function_call} \rangle \rightarrow \text{id} (\langle \text{args} \rangle) ;$
32	$\langle \text{args} \rangle \rightarrow \epsilon$

N° RP	Règle de production
33	$\langle \text{args} \rangle \rightarrow \text{string } \langle \text{separateur} \rangle \langle \text{args} \rangle$
34	$\langle \text{args} \rangle \rightarrow \langle \text{operand} \rangle \langle \text{separateur} \rangle \langle \text{args} \rangle$
35	$\langle \text{separateur} \rangle \rightarrow ,$
36	$\langle \text{separateur} \rangle \rightarrow \epsilon$

code source:

```
static const int RPs[][][10] = {
    /* 0 */ { CODE_DO, CODE_ACCLADE_OUVRANTE, NT_STATEMENTS,
CODE_ACCLADE_FERMANTE,
            CODE_WHILE, CODE_PARENTHES_OUVRANTE, NT_CONDITION,
CODE_PARENTHES_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
<statements>	er	er	1	1	1	Er	2	2	er	er	er	er	1	1	er	er	er	er										
<statement>	er	er	31	5	5	Er	3	4	er	er	3	4	er	er	er	er	er	er	er	er	er							
<declare_stmt>	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
<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
<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	er	er	er	er
<operation>	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
<increment_stmt>	17	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
<decrement_stmt>	18	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
<condition>	er	er	11	er	er	Er	er	er	19	20	21	22	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
<bool_op>	er	er	er	er	er	Er	er	er	er	er	er	er	er	27	28	er	er	er	er									
<appelfonction>	er	er	31	er	er	Er	er	er	er	er	er	er	er	er	er	er	er	er	er	er	er	er	33	er	er	er	er	er
<args>	er	er	34	er	er	Er	er	er	32	32	er	34	33	er	er	er	er	er										
<separateur>	er	er	er	er	er	Er	er	er	36	er	35	er	er	er	er	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

The screenshot shows the VS Code interface with the following details:

- File Explorer:** Shows files like build.bat, README.md, code_source.txt (marked with an error icon), lexime.c, lexime.h, analyseur_lexical.h, analyseur_syntactique.c, analyseur_syntactique.h, main.c, msn.exe, grammaire.txt, and grammaire.h.
- Editor:** Displays the content of code_source.txt with several syntax errors highlighted in red. The code includes a do-while loop and various C library calls.
- Terminal:** Shows the command line output of the compiler, listing numerous lexical errors (Lexeme: [] Code: [] Ligne: []) and a stack overflow error (40 pile 102 suite ul 105).
- Status Bar:** Provides information about the file path (C:\Users\...), current status (Not Committed Yet), and date (07-May-25).

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
```

The screenshot shows the Visual Studio Code interface with the following details:

- File Explorer:** Shows files like `build.bat`, `README.md`, `code_source.txt`, `lexime.c`, `lexime.h`, `analyseur_lexical.h`, `analyseur_syntaxique.c`, `analyseur_syntaxique.h`, `main.c`, and `msn.exe`.
- Terminal:** The title bar says "msn-compiler". The terminal content is:

```
96 pile 105 suite ul 105
-----PTILE-----
pile: Lexeme: [#] Code: [999] Ligne: [-2]
-----SUIT UL-----suiteUL:
Lexeme: [#] Code: [999] Ligne: [10]

Analyse syntaxique terminée avec succès.
La structure do-while est syntaxiquement correcte.
```
- Status Bar:** Shows "Not Committed Yet", "Ln 9, Col 13", "Spaces: 4", "UTF-8", "CRLF", "Plain Text", "Go Live", the date "07-May-25", and the time "7:04 PM".