

Rapport Projet Compilation

Première

e

Année d'Ingénieur :
1ING4

Spécialité : IDISC-ISEOC

Développement d'un interpréteur acceptant les requêtes de manipulation et d'accès à une base de données

Réalisé par :

Iheb Aouini

Walid Ben Abi Taieb

Aya Soltani

Youssef Jedidi

Année Universitaire :
2023-2024

Table des matières

1	Introduction	2
2	Concepts de base	2
2.1	Analyseur lexical.....	2
2.2	Analyseur Syntaxique.....	2
3	Les outils de travail	3
3.1	Flex.....	3
3.2	Bison.....	3
4	Réalisation	5
4.1	Partie 1 : Développement de l'analyse lexicale :.....	5
4.1.1	-Les déclarations en C :.....	5
4.1.2	-Les unités lexicales :.....	5
4.1.3	Les opérandes et les caractères spécifiques :.....	6
4.1.4	Variables à afficher :.....	7
4.2	Partie 2 : Développement de l'analyse syntaxique.....	7
4.2.1	-Les déclarations en C :.....	7
4.2.2	-Les déclarations des unités lexicales.....	8
4.2.3	Définition de la grammaire :.....	8
4.2.4	-Bloc principal et fonctions auxiliaires en C :.....	10
5	Guide d'utilisation	10
6	Conclusion	13

1 Introduction

- ★ Un compilateur prend un programme écrit dans un langage de haut niveau et le traduit en une séquence d'instructions élémentaires que la machine peut exécuter. La création de compilateurs a longtemps été une activité centrale en programmation, conduisant au développement de nombreuses techniques et théories. La compilation d'un programme se déroule en trois phases :
- ★ La première phase, l'analyse lexicale, découpe le programme en unités lexicales telles que les opérateurs, les mots réservés, les variables et les constantes.
- ★ La deuxième phase, l'analyse syntaxique, représente la structure du programme sous forme d'un arbre syntaxique où chaque nœud correspond à un opérateur et ses fils aux opérandes sur lesquels il agit.
- ★ La troisième phase vérifie la sémantique du langage de programmation et, enfin, la phase de génération de code construit la suite d'instructions du processeur à partir de l'arbre syntaxique.

2 Concepts de base

2.1 Analyseur lexical

L'analyse lexicale est la première phase de la compilation. Dans le texte source, qui se présente comme un flot de caractères, l'analyse lexicale reconnaît des unités lexicales, qui sont les mots avec lesquels les phrases sont formées, et les présente à la phase suivante, l'analyse syntaxique.

Les principales sortes d'unités lexicales qu'on trouve dans les langages de programmation courants sont :

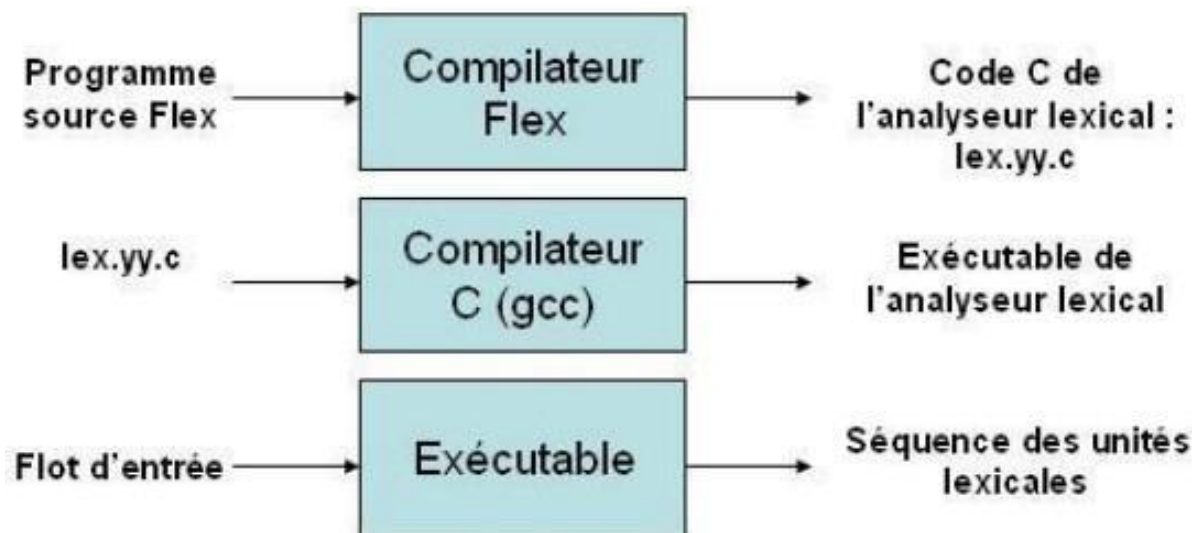
- Les caractères spéciaux simples : +, =, etc. ;
- Les caractères spéciaux doubles : <=, ++, etc. ;
- Les mots-clés : if, while, etc. ;
- Les constantes littérales : 123, -5, etc. ;
- Les identificateurs : i, vitesse1, etc

2.2 Analyseur Syntaxique

L'analyse syntaxique implique la segmentation d'un énoncé en langage informatique en plusieurs composantes interprétables par l'ordinateur. Dans le cadre d'un compilateur, un analyseur syntaxique est un programme chargé de décomposer chaque déclaration écrite par un développeur en éléments distincts tels que la commande principale, les options, les objets cibles et leurs attributs. Ces éléments peuvent ensuite être utilisés pour définir d'autres actions ou pour composer des instructions constituant un programme exécutable. Le rôle de l'analyseur syntaxique comprend :

2 Concepts de base

- Assurer la conformité d'une construction à une syntaxe et à une grammaire spécifiées.
 - Valider la séquence d'unités lexicales en les analysant de gauche à droite.
 - Organiser les unités lexicales en structures grammaticales telles que des déclarations, des affectations ou des appels de fonction.
 - Créer un arbre syntaxique représentant les unités lexicales identifiées
- ❖ Les opérateurs sont les nœuds, les opérandes sont les fils du nœuds correspondant à cet opérateur



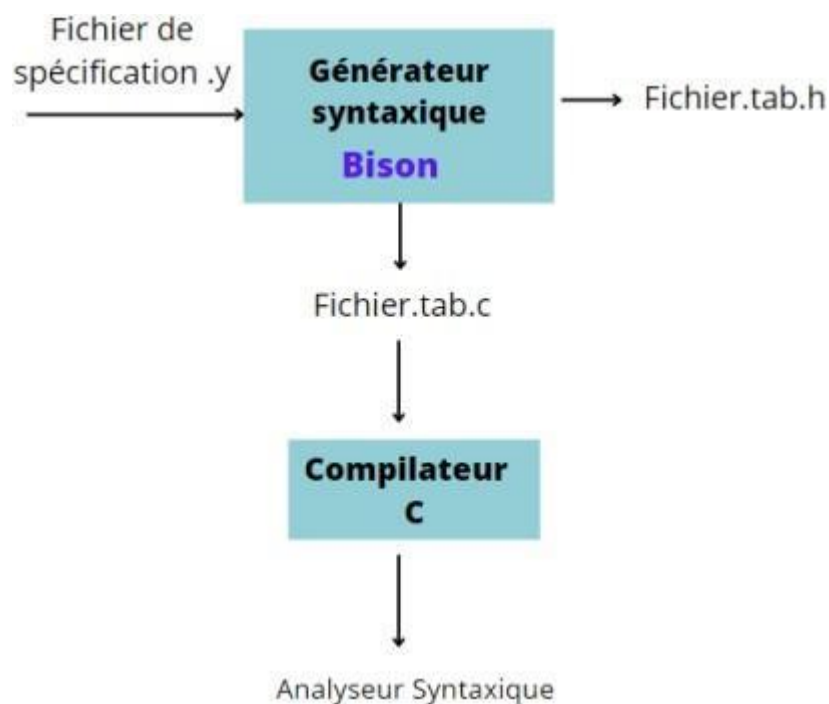
3 Les outils de travail

3.1 Flex

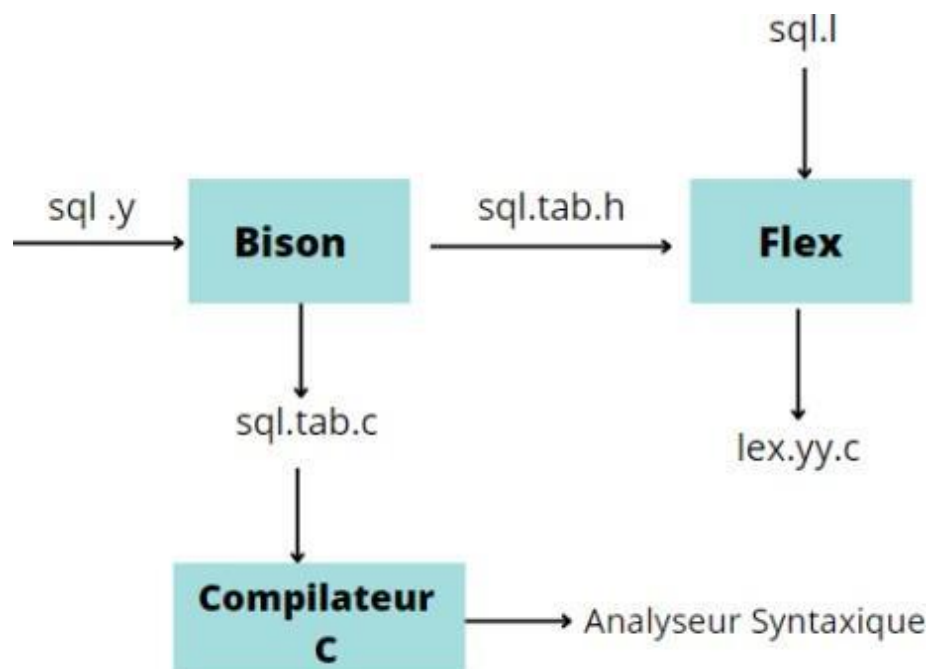
Flex est une alternative open source à l'analyseur lexical Lex, souvent utilisée conjointement avec l'analyseur syntaxique GNU Bison, équivalent de Yacc. Il s'agit d'un outil de génération d'analyseurs syntaxiques qui reconnaissent les motifs lexicaux dans un texte. Flex accepte des expressions régulières en entrée, représentant les unités lexicales, et génère un programme en langage C qui, une fois compilé, peut reconnaître ces unités lexicales. Le fichier source C généré, appelé 'lex.yy.c', contient une fonction 'yylex()' qui est ensuite compilée et liée avec l'option '-lfl' (correspondant à la bibliothèque flex) pour créer un exécutable. Lorsque cet exécutable est lancé, il analyse son entrée à la recherche de correspondances avec les expressions régulières précédemment définies. Pour chaque expression trouvée, le code C correspondant est exécuté.

3.2 Bison

Bison, la version GNU de YACC, est un utilitaire permettant la génération automatique d'analyseurs syntaxiques. Il prend en entrée la description d'un langage sous forme de grammaire et produit un programme en langage C qui, une fois compilé, est capable de reconnaître les mots ou les programmes conformes à la grammaire spécifiée. Un analyseur syntaxique créé avec Bison est essentiellement une représentation de la grammaire d'un langage. Les programmes Bison sont simplement des fichiers texte enregistrés avec l'extension ".y".



La génération d'analyseur syntaxique à l'aide de Flex et Bison est illustrée par la figure suivante :



4 Réalisation

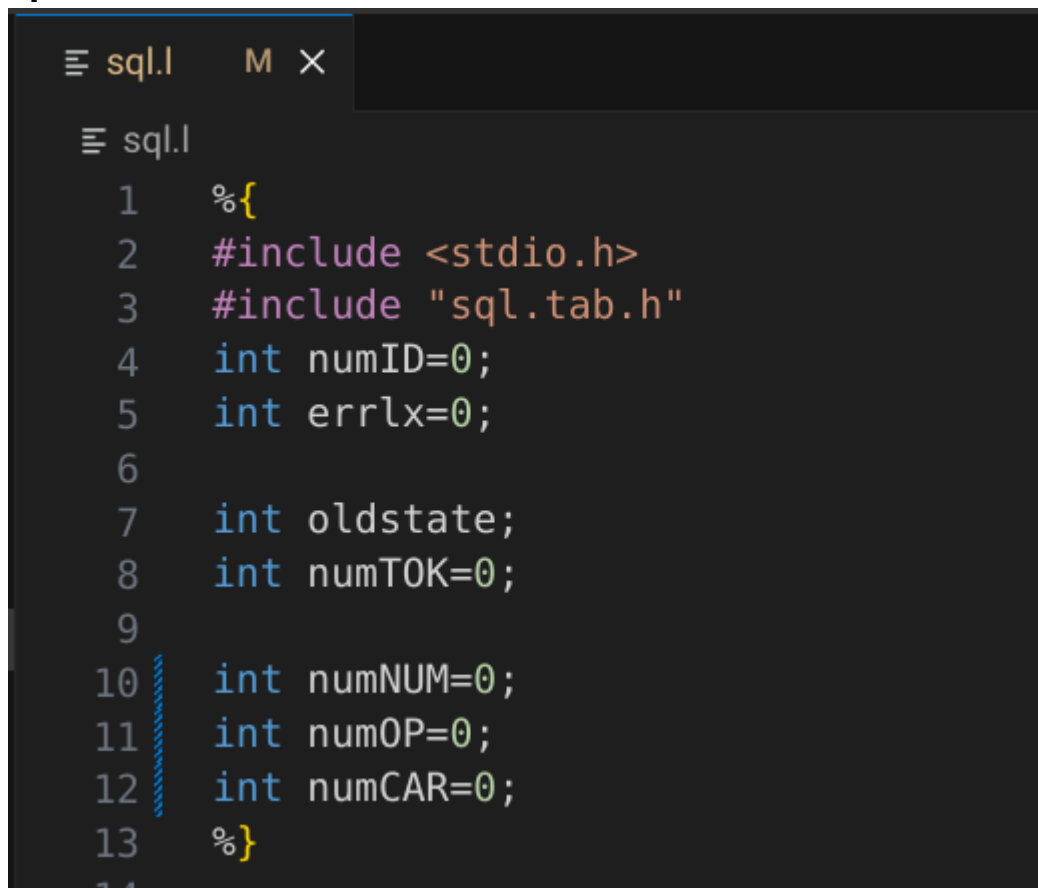
Nous nous engageons dans un projet visant à créer un interpréteur capable de traiter les demandes de manipulation et d'accès à une base de données. Les requêtes à prendre en charge par cet interpréteur sont définies comme suit :

- Langage de Manipulation de Données (LMD) : Create / delete / update
- Langage d'Interrogation des Données (LID) : Select

4.1 Partie 1 : Développement de l'analyse lexicale :

Dans cette partie nous allons définir le fichier de spécification Flex qui introduit notre analyseur lexicale « sql.l » comme suit :

4.1.1 -Les déclarations en C :



```
≡ sql.l  M X
≡ sql.l
1  %{
2  #include <stdio.h>
3  #include "sql.tab.h"
4  int numID=0;
5  int errlx=0;
6
7  int oldstate;
8  int numTOK=0;
9
10 // int numNUM=0;
11 // int numOP=0;
12 // int numCAR=0;
13 %}
```

4.1.2 -Les unités lexicales :

Nous avons introduit notre analyseur lexical qui présente tous les mots prédéfinis, nécessaires et existants dans le langage SQL.

4.1.3 Les opérandes et les caractères spécifiques :

Nous présentons ainsi la définition des opérandes et des caractères spécifiques comme le montre la figure suivante.

```
37 LESSOrEQUAL (" $\leq$ ")
38 GREATEROrEQUAL (" $\geq$ ")
39 DISTINCT (" $\neq$ ")
40 GREATER \ $\gt$ 
41 LESSER \ $\lt$ 
42 EQUAL \ $\equiv$ 
43 PAR_OUV \ $\left($ 
44 PAR_FER \ $\right)$ 
45 PT_VIR \ $\;$ 
46 VIR \ $\,$ 
47 PT \ $\.$ 
48 COTE \ $\prime$ 
49 DOUBLECOTE \ $\prime\prime$ 
50 ETOILE \ $\ast$ 
51 CHIFFRE  $[0-9]^+$ 
52 ID  $[A-Za-z][A-Za-z0-9]^*$ 
53 IGNORE (" $\backslash$ " " $\backslash t$ " " $\backslash n$ ")*
```

```

56 %%
57 {QUIT} {printf("\n\nLa phase d'analyse lexical est terminé -> Resultat de l'analyseur lexical : \n\t -
58 <<EOF>> {
59     printf("\nEnd of file reached.\n");
60
61     return 0;
62 }
63 {PRINT} {printf("\n Nombre des champs : %d ", numID); return 0;}
64 {AND} {printf(" AND %s\n", yytext); ++numTOK; return AND;}
65 {OR} {printf(" OR: %s\n", yytext); ++numTOK; return OR;}
66 {PAR_OUV} {printf(" PAR_OUV: %s\n", yytext); ++numTOK; ++numCAR; return PAR_OUV;}
67 {PAR_FER} {printf(" PAR_FER: %s\n", yytext); ++numTOK; ++numCAR; return PAR_FER;}
68 {PT_VIR} {printf(" PT_VIR: %s \n==> SQL requete , line %d : ", yytext,yylineno); ++numTOK; ++numCAR
69 {VIR} {printf(" VIR: %s\n", yytext); ++numTOK; ++numCAR; return VIR;}
70 {PT} {printf(" PT: %s\n", yytext); ++numTOK; return PT;}
71 {COTE} {printf(" COTE: %s\n", yytext); ++numTOK; ++numCAR; return COTE;}
72 {DOUBLECOTE} {printf(" DOUBLECOTE: %s\n", yytext); ++numTOK; ++numCAR; return DOUBLECOTE;}
73 {ETOILE} {printf(" ETOILE: %s\n", yytext); ++numTOK; ++numCAR; return ETOILE;}
74 {GREATERorEQUAL} {printf(" GREATERorEQUAL: %s\n", yytext); ++numTOK; ++numOP; return GREATERorEQUAL;}
75 {LESSorEQUAL} {printf(" LESSorEQUAL: %s\n", yytext); ++numTOK; ++numOP; return LESSorEQUAL;}
76 {DISTINCT} {printf(" DISTINCT: %s\n", yytext); ++numTOK; ++numOP; return DISTINCT;}
77 {LESSER} {printf(" LESSER: %s\n", yytext); ++numTOK; ++numOP; return LESSER;}
78 {GREATER} {printf(" GREATER: %s\n", yytext); ++numTOK; ++numOP; return GREATER;}
79 {EQUAL} {printf(" EQUAL: %s\n", yytext); ++numTOK; ++numOP; return EQUAL;}
80 {CREATE} {printf(" CREATE: %s\n", yytext); ++numTOK; return CREATE;}
81 {TABLE} {printf(" TABLE: %s\n", yytext); ++numTOK; return TABLE;}
82 {SELECT} {printf(" SELECT: %s\n", yytext); ++numTOK; return SELECT;}
83 {SET} {printf(" SET: %s\n", yytext); ++numTOK; return SET;}
84 {UPDATE} {printf(" UPDATE: %s\n", yytext); ++numTOK; return UPDATE;}
85 {DELETE} {printf(" DELETE: %s\n", yytext); ++numTOK; return DELETE;}

```

Pour savoir la longueur de l'unité lexicale contenue dans yytext de n'importe quel mot de langage SQL, nous avons bien utilisé la variable yylen et nous avons pu l'afficher avec un printf. Pour savoir la longueur de l'unité lexicale contenue dans yytext de n'importe quel mot de langage SQL, nous avons bien utilisé la variable yylen et nous avons pu l'afficher avec un printf.

4.1.4 Variables à afficher :

Initialisation du nombre des tokens , des champs et des erreurs, ainsi que le nombre des valeurs numérique , caractère et des opérateurs,

```

4   int numID=0;
5   int errlx=0;
6
7   int oldstate;
8   int numTOK=0;
9
10  int numNUM=0;
11  int numOP=0;
12  int numCAR=0;

```

Nous avons compilé ce fichier avec flex pour générer le fichier lex.yy.c. avec la commande : flex sql.l

4.2 Partie 2 : Développement de l'analyse syntaxique :

Nous passons maintenant à la partie déclarative de notre grammaire dans notre fichier bison sql.y qui représente notre analyseur syntaxique.

Ce fichier.y est composé par 4 parties :

4.2.1 -Les déclarations en C :

```
1  %{
2  #include <math.h>
3  #include <stdio.h>
4  #include <stdlib.h>
5  #include "lex.yy.c"
6  int yylex();
7  int yyerror();
8  int nb_champ=1;
9  int is_select_all=0;
10 %}
```

nous commençons par inclure le fichier lex.yy.c, qui est généré par Flex et contient le code source du scanner lexical. Ensuite, nous déclarons deux fonctions : yylex() et yyerror(), utilisées respectivement pour l'analyse du flux d'entrée et la gestion des erreurs lors de l'analyse syntaxique. De plus, nous initialisons deux variables : nb_champ, qui est attribuée à 1 cet champ est un compteur de nombre de champ, et is_select_all, initialisée à 0 cet champ est mis à 1 si et seulement si on a un select * cad selection de tous les champs .

Enfin, la directive %} marque la fin du bloc de code encapsulé par %{ et %}, où sont définies ces variables et fonctions. Ces éléments sont essentiels pour la configuration et le bon fonctionnement de l'analyseur lexical Flex.

4.2.2 -Les déclarations des unités lexicales :

La fonction `int yylex()` est déclarée pour la lecture des tokens (fournie par Flex). Nous avons passé maintenant à la déclaration des tokens.

```
19 //les Tokens
20 %token AND
21 %token OR
22 %token LESSorEQUAL
23 %token GREATERorEQUAL
24 %token DISTINCT
25 %token GREATER
26 %token LESSER
27 %token EQUAL
28 %token CREATE
29 %token TABLE
30 %token SELECT
31 %token UPDATE
32 %token SET
33
34 %token DELETE
35 %token INSERT
36 %token INTO
37 %token FROM
38 %token WHERE
39 %token GROUP
40 %token ORDER
```

4.2.3 Définition de la grammaire :

Pour établir une grammaire qui débute avec la directive `%start grammaire`, nous avons élaboré la structure en commençant par l'axiome, qui représente la règle initiale de la grammaire, puis nous avons spécifié les règles de production pour chaque type de requête SQL.

```

%%
grammaire : Axiome;
Axiome  : requete PT_VIR loop | QUIT PT_VIR | QUIT;
loop    : Axiome | /*epsilon*/ ;
requete : creer | selectionner | delete | insertion | update ;
creer   : CREATE TABLE newTable {printf(" Creation de la table \n\n");};
newTable : ID PAR_OUV Colonne PAR_FER | ID ;
selectionner : SELECT id FROM table options{printf(" Affichage de(s) ligne(s) de la table ") ; if (is_select_all==0) {printf("\n Non ");}};
options  : opt1 | opt2 | opt3 | /*epsilon*/ ;
opt1     : WHERE exp opt4 | /*epsilon*/;
opt2     : ORDER BY table ASC  opt5 | ORDER BY table DESC opt5;
opt3     : GROUP BY ID;
opt4     : opt2 | opt3 | /*epsilon*/;
opt5     : opt3 | /*epsilon*/;
delete   : DELETE FROM table opt1 { printf("Suppression de(s) ligne(s) de la table \n\n");};

update   : UPDATE table SET Colonne operateur type WHERE Colonne operateur type {printf("Mise a jour effectué avec succes \n\n");};

insertion : INSERT INTO table VALUES PAR_OUV ident1 PAR_FER {printf("Insertion d'une ligne dans la table \n\n");};
ident1    : ident2 | COTE text COTE | NUM | DOUBLECOTE text DOUBLECOTE ;
ident2    : ID | ID_PT ID;

```

L'axiome est formulé comme une requête suivie d'un point-virgule et d'une boucle, permettant ainsi la récursivité de la grammaire pour autoriser des requêtes multiples. Les requêtes possibles comprennent la création, la sélection, la suppression et l'insertion. Chaque type de requête est caractérisé par une règle de production décrivant sa structure en termes de mots-clés et de paramètres. D'autres règles de production spécifient les différents paramètres envisageables pour chaque type de requête. Par exemple, la règle 'options' établit les options possibles pour une requête de sélection, telles que le tri des résultats ou le regroupement des données. Les règles 'opt1', 'opt2' et 'opt3' détaillent les différentes options de tri ou de regroupement envisageables.

Afin d'afficher les messages d'erreurs syntaxique avec la ligne d'erreur dans le script (input file) , nous avons utilisé la fonction **int yyerror(char const *)**.

```
98
99 int yyerror(const char *str)
100 {
101
102     fprintf(stderr,"Erreur syntaxique | Ligne: %d\n%s\n",yylineno,str);
103
104     int a;
105     scanf ("%d",&a);
106 }
107
108
```

4.2.4 -Bloc principal et fonctions auxiliaires en C :

Pour vérifier l'existence d'un fichier passé en paramètre on a créé une fonction main () qui affiche une erreur dans le cas contraire.

```
109 int main (int argc, char *argv[])
110 {++argv,--argc;
111 printf("Debut de l'analyse\n\n");
112
113 yyparse();
114 printf("");
115 getchar();
116 printf("\n\nFin de l'analyse \n \n");
117
118 freopen("/dev/tty", "r", stdin); // to redirect input to the terminal
119
120 char str[5];
121 printf("Do you want to scan another sql script [Y|N] : ");
122 scanf("%s", str);
123 if (str[0]=='Y'){
124     printf("\033[2J"); // Clear the screen using ANSI escape sequence
125     printf("\033[1;1H"); // Move cursor to top-left corner
126
127     char filename[100];
128     printf("\n Enter filename of the sql script : ");
129     scanf("%s", filename);
130     char command[115];
131     sprintf(command, "./sql < %s", filename);
132     system(command);
133 }
134 else {
135     printf("\n Bye \n");
136 }
```

Pour compiler le fichier de spécification syntaxique bison nous avons commencé par taper la commande **bison -d sql.y**.

Deux fichiers seront générés **sql.tab.c** et **sql.tab.h**.

Enfin, nous avons introduit le fichier `lex.yy.c` dans le fichier de spécification de Bison, ainsi que le fichier `sql.tab.h` dans le fichier de spécification de Flex.

5 Guide d'utilisation

- Nous avons créé un script shell `'compile_me.sh'` pour compiler les fichiers sources automatiquement, voici le contenu du script `compile_me`

```
$ compile_me.sh M X
$ compile_me.sh
1  bison -d sql.y ;
2  flex sql.l;
3  gcc sql.tab.c -o sql ;
4
5  #executing the project :  ./projet < test.txt
6
7
```

pour compiler, dans la terminal :

```
leadar@ti44:~/Downloads/Compilation_sql_parser
~/Dow/Compilation_sql_parser > main !7 ./compile_me.sh
sql.y: warning: 1 reduce/reduce conflict [-Wconflicts-rr]
sql.y: note: rerun with option '-Wcounterexamples' to generate conflict counterexamples

~/Dow/Compilation_sql_parser > main !7 ls
clean.sh      lex.yy.c      README.md    sql.l        sql.tab.h    test.txt
compile_me.sh Rapport_Projet_compilation.pdf sql          sql.tab.c    sql.y

~/Dow/Compilation_sql_parser > main !7
```

pour exécuter : on exécute et passe le fichier input `test.txt` qui contient les requêtes

```
~/Dow/Compilation_sql_parser > main !3 ./sql < test.txt
Debut de l'analyse

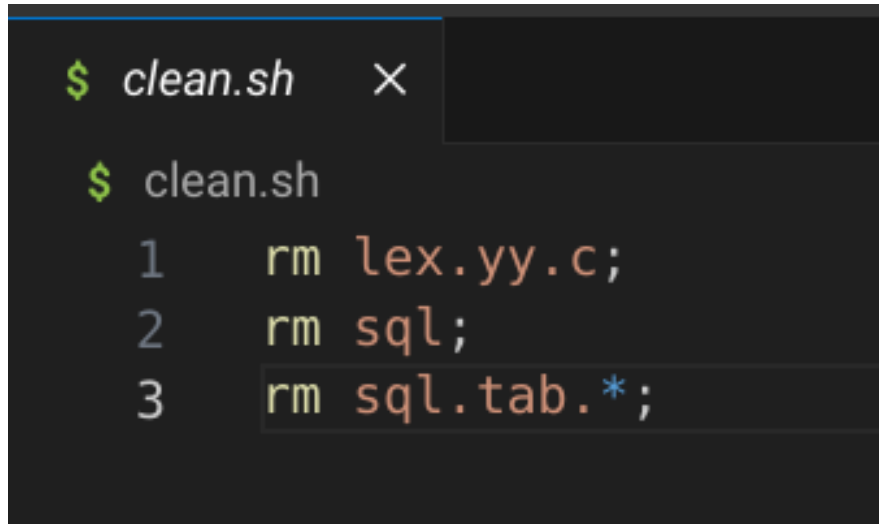
SELECT: SELECT
ETOILE: *
FROM: from
ID: users
PT_VIR: ;
==> SQL requete , line 1 :  Affichage de(s) ligne(s) de la table
Nombre de champ de cet requete :Selection de Tous les champs (Etoile)
```

Nous avons aussi un script shell 'clean.sh' pour supprimer les fichier compilé (les binaires)

pour l'executer :

chmod +x clean.sh ;

./clean.sh

A terminal window with a dark background. The prompt is a green '\$'. The command 'clean.sh' is entered and executed. The output shows three lines of commands: '1 rm lex.yy.c;', '2 rm sql;', and '3 rm sql.tab.*;'. The third line is highlighted with a light blue selection box.

```
$ clean.sh
1  rm lex.yy.c;
2  rm sql;
3  rm sql.tab.*;
```

Exécution et testing à partir de l'invite de commande :

```
+ ./sql < test.txt
~/Dow/Compilation_sql_parser > main !4 ./sql < test.txt
Debut de l'analyse

SELECT: SELECT
ID: nom
VIR: ,
ID: age
FROM: from
ID: clients
PT_VIR: ;
==> SQL requete , line 1 : Affichage de(s) ligne(s) de la table
Nombre de champ de cet requete : 2

CREATE: CREATE
TABLE: TABLE
ID: tt
PT_VIR: ;
==> SQL requete , line 2 : Creation de la table

DELETE: DELETE
FROM: FROM
ID: tab
WHERE: where
```

Pour afficher une erreur dans une ligne au niveau de l'input, on tape les fausse requêtes sql suivantes : **SELECT nom,id,salaire from ;** et **SELECT num ;**

```
+ ./sql < test.txt
~/Dow/Compilation_sql_parser > main !7 ./sql < test.txt
Debut de l'analyse

SELECT: SELECT
ID: nom
VIR: ,
ID: id
VIR: ,
ID: salaire
FROM: from
PT_VIR: ;
Erreur syntaxique | Ligne: 1
syntax error
```

```
~/Dow/Compilation_sql_parser > main !7 ./sql < test.txt
Debut de l'analyse

SELECT: SELECT
ID: num
PT_VIR: ;
Erreur syntaxique | Ligne: 1
syntax error
```

NB : On constate une erreur syntaxique dans la ligne 1 pour la première requête et le champ ID : fro dans la ligne 1 pour la deuxième requête.

SELECT nu/m fro pers ;

```
~/Dow/Compilation_sql_parser > main !7 ./sql < test.txt
Debut de l'analyse

SELECT: SELECT
ID: nu
Erreur lexicale | Ligne: 1 , What do you mean by : '/' !
```

Nous constatons une erreur lexicale dans le colonne nu/m (on peut pas utilisé le / dans les id) : **SELECT nu/m fro pers ;** dans la ligne 1

Nous avons introduit cette commande **./sql < test.txt** pour exécuter le fichier contenant les requêtes.

```
test.txt M x
test.txt
1  SELECT * from users;
2  DELETE FROM tab where a=2;
3  SELECT id,nom;
4  INSERT INTO users VALUES ('walid');
5  UPDATE user SET salaire=1500 WHERE id=5;
6
7  QUIT
```



```

~/Dow/Compilation_sql_parser > main !7 ./sql < test.txt
Debut de l'analyse

SELECT: SELECT
ETOILE: *
FROM: from
ID: users
PT_VIR: ;
==> SQL requete , line 1 : Affichage de(s) ligne(s) de la table

DELETE: DELETE
FROM: FROM
ID: tab
WHERE: where
ID: a
EQUAL: =
NUM: 2
PT_VIR: ;
==> SQL requete , line 2 : Suppression de(s) ligne(s) de la table

SELECT: SELECT
ID: id
VIR: ,
ID: nom
PT_VIR: ;
Erreur syntaxique | Ligne: 3
syntax error

```

Calcul des champs :

on a ajouter des variable et des action dans le fichier sql.y pour le calcul de nombre de champ

```

SELECT: SELECT
ID: nom
VIR: ,
ID: age
FROM: from
ID: clients
PT_VIR: ;
==> SQL requete , line 1 : Affichage de(s) ligne(s) de la table
Nombre de champ de cet requete : 2

```

grace a cet modification dans la grammaire :

```

selectionner : SELECT id FROM table
options{printf(" Affichage de(s) ligne(s) de la table ") ; if
(is_select_all==0) {printf("\n Nombre de champ de cet requete :
%d\n\n",nb_champ) ;} else {printf("\n Nombre de champ de cet requete
:Selection de Tous les champs (Etoile) \n\n") ;}
nb_champ=1;is_select_all=0;};

```

Statistique de l'analyse lexical :

avec Nous avons aussi ajouter une parti de statistique :

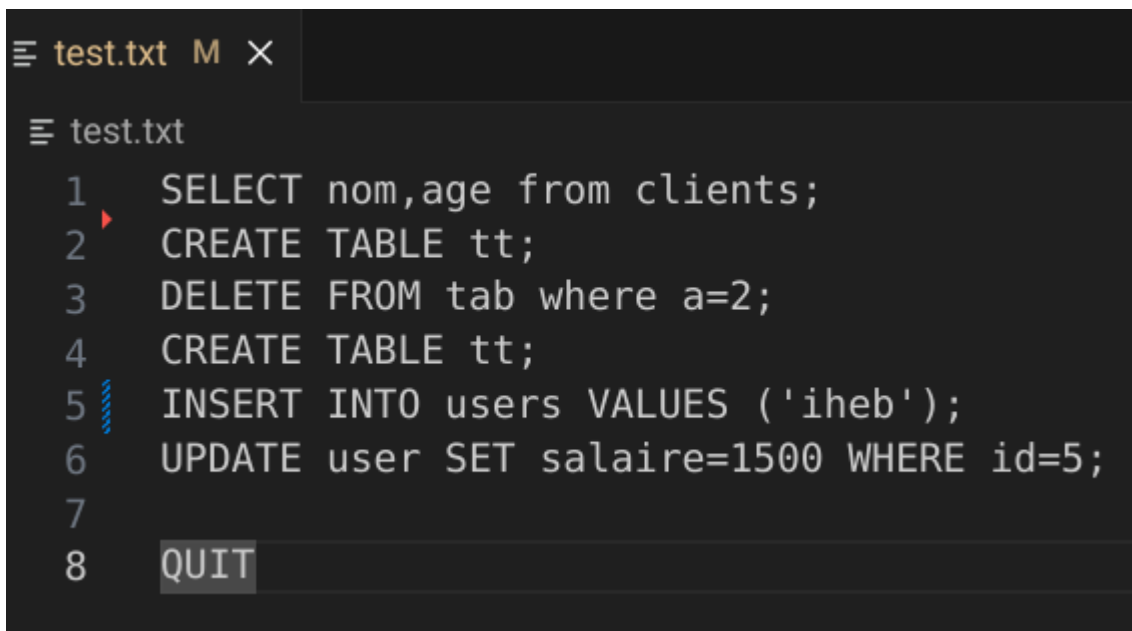
```
La phase d'analyse lexical est terminé -> Resultat de l'analyseur lexical :  
  - Nombres d'erreurs Lexical(Characteres inconnus ) : 0  
  - Mots cle: 42  
  - IDs: 10  
  - Numeros: 3  
  - Operateurs: 3  
  - Caracteres speciaux(*,;): 11  
  
End of file reached.  
  
Fin de l'analyse  
  
Do you want to scan another sql script [Y|N] : █
```

L'analyse lexicale est une étape préliminaire dans le traitement de texte où le programme examine le texte pour identifier différents éléments comme les mots-clés, les identificateurs, les nombres, les opérateurs et les caractères spéciaux. Dans ce cas précis, cette phase d'analyse a été menée à terme, et voici les résultats obtenus :

- Aucune erreur lexicale n'a été détectée, ce qui signifie qu'aucun caractère inconnu n'a été rencontré.
- Le texte contient 42 mots-clés.
- Il y a 10 identificateurs, probablement des noms de variables ou de fonctions.
- Trois nombres ont été identifiés dans le texte.
- Trois opérateurs ont été repérés, comme des symboles mathématiques ou logiques.
- On a relevé 11 caractères spéciaux, tels que des astérisques (*) ou des points-virgules (;).

Après cette analyse, le programme a atteint la fin du fichier et a terminé son processus d'analyse lexicale.

Pour que la statistique s'exécute il faut mettre le token QUIT dans la fin de fichier d'input qui contient les requets :



```

test.txt M X
test.txt
1  SELECT nom,age from clients;
2  CREATE TABLE tt;
3  DELETE FROM tab where a=2;
4  CREATE TABLE tt;
5  INSERT INTO users VALUES ('iheb');
6  UPDATE user SET salaire=1500 WHERE id=5;
7
8  QUIT

```

Autre utilité (re scanner ou quitter)

On a aussi ajouter le feature et la possibilité de re scanner un autre fichier sql (input) on choisissant Y , ensuite en tape le nom de fichier désiré , ou N pour quitter le programme

Y:

```
Enter filename of the sql script : test.txt
```

```
Do you want to scan another sql script [Y|N] : Y
```

```
Enter filename of the sql script : test.txt
```

```
Debut de l'analyse
```

```
SELECT: SELECT
```

```
ETOILE: *
```

```
FROM: from
```

```
ID: users
```

```
PT_VIR: ;
```

```
==> SQL requete , line 1 : Affichage de(s) ligne(s) de la table
```

```
CREATE: CREATE
```

N :

```
Fin de l'analyse
```

```
Do you want to scan another sql script [Y|N] : N
```

```
Bye
```

```
~/Downloads/Compilation_sql_parser > main !7
```

6 Conclusion

Ce code illustre l'implémentation d'un analyseur syntaxique pour le langage SQL en utilisant Flex pour la génération du lexer et en définissant les règles de production de la grammaire avec la syntaxe de Bison. Les bibliothèques standard telles que math.h, stdio.h et stdlib.h sont également utilisées pour les fonctions mathématiques et les opérations d'E/S. L'objectif principal de ce programme est de reconnaître et de valider la syntaxe des requêtes SQL saisies par l'utilisateur ou lues à partir d'un fichier. En cas d'erreur, le programme affiche un message d'erreur accompagné du numéro de ligne correspondant.