

RAPPORT DE MINI PROJET COMPILATION

PREMIÈRE ANNÉE CYCLE D'INGÉNIEUR

Spécialité : ISEOC

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

Par

CHAIMA JOUINI
INSAF LOUSSAIEF
NOUR DHRIF
RANIA LEJMI

Année Universitaire : 2022-2023

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 transforme un programme écrit en langage évolué en une suite d'instructions élémentaires exécutables par une machine. La construction de compilateurs a longtemps été considérée comme une des activités fondamentales en programmation, elle a suscité le développement de très nombreuses techniques qui ont aussi donné lieu à des théories maintenant classiques. La compilation d'un programme est réalisée en trois phases, la première, analyse lexicale, consiste à découper le programme en petites entités : opérateurs, mots réservés, variables, constantes numériques, alphabétiques, etc. La deuxième phase, analyse syntaxique, consiste à expliciter la structure du programme sous forme d'un arbre, appelé arbre de syntaxe, chaque nœud de cet arbre correspond à un opérateur et ses fils aux opérandes sur lesquels il agit. La troisième phase, qui effectue les vérifications nécessaires à la sémantique du langage de programmation considéré et finalement, la phase de génération de code, construit la suite d'instructions du micro-processeur à partir de l'arbre de syntaxe.

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 :

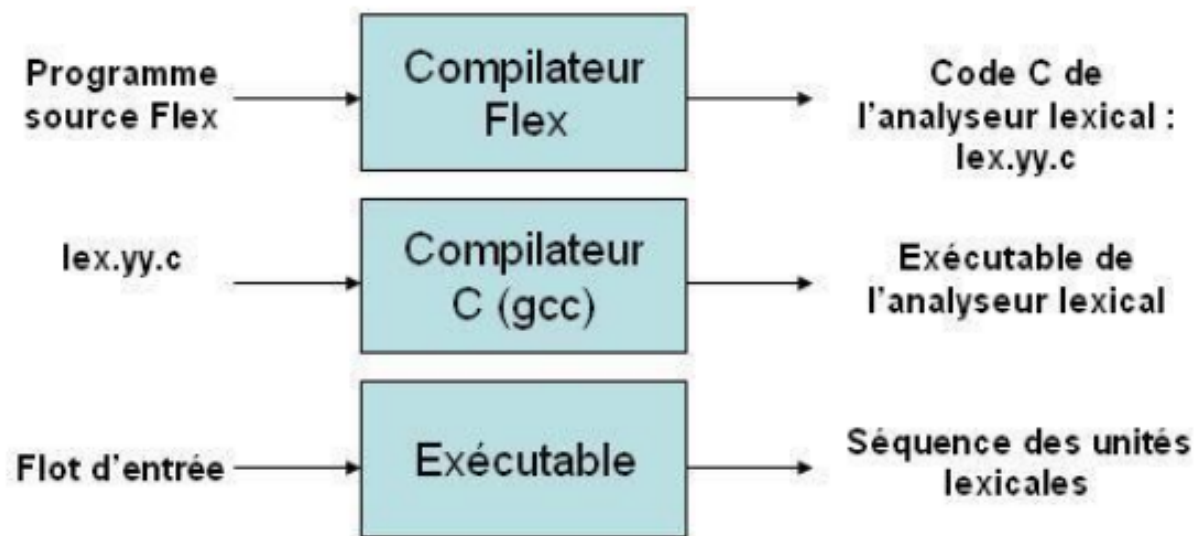
- Les caractères spéciaux simples : +, =, etc. ;
- Les caractères spéciaux doubles : <=, ++, etc. ;
- Les mots-clés : if, while, etc. ;
- Les constantes littérales : 123, -5, etc. ;
- Les identificateurs : i, vitesse1, etc

2.2 Analyseur Syntaxique

L'analyse syntaxique consiste à diviser un énoncé en langage informatique en plusieurs parties exploitables par l'ordinateur. Dans un compilateur, un analyseur syntaxique est un programme qui prend chaque déclaration écrite par un développeur et la découpe en morceaux (par exemple, la commande principale, les options, les objets cibles, leurs attributs, etc.). Ces morceaux peuvent ensuite être utilisés pour développer d'autres actions ou créer des instructions formant un programme exécutable. Le rôle de l'analyseur syntaxique consiste à :

- Vérifier si une construction respecte une syntaxe et une grammaire bien définie.
- Vérifier que la suite d'unités lexicales en les parcourant de gauche à droite est correcte.
- Regrouper les unités lexicales en structure grammaticales (déclaration, affectation, appel de fonction...)
- Construction d'un arbre syntaxique qui représente les unités lexicales obtenues.

- Les opérateurs sont les nœuds, les opérandes sont les fils du nœuds correspondant à cet opérateur



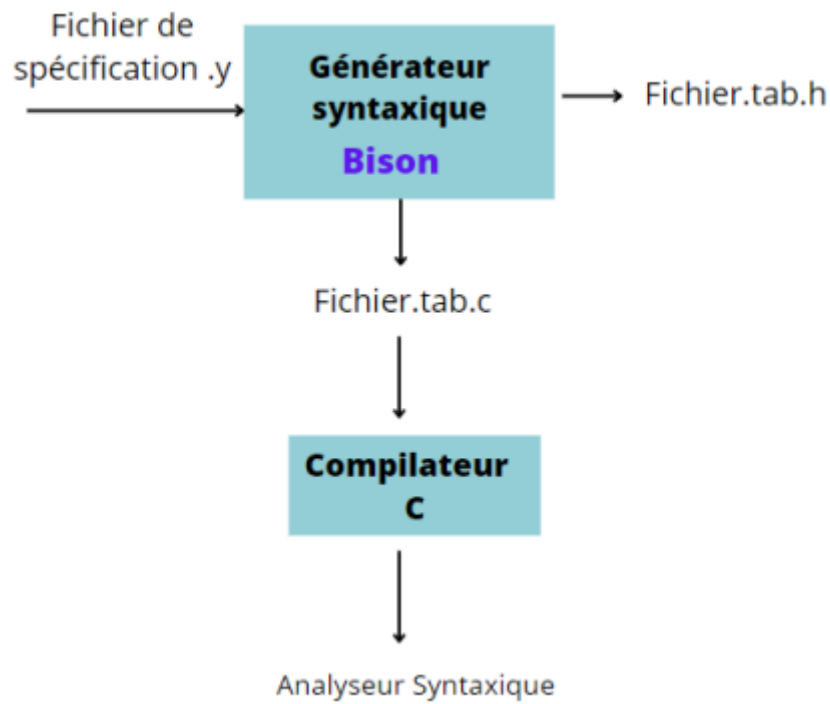
3 Les outils de travail

3.1 Flex

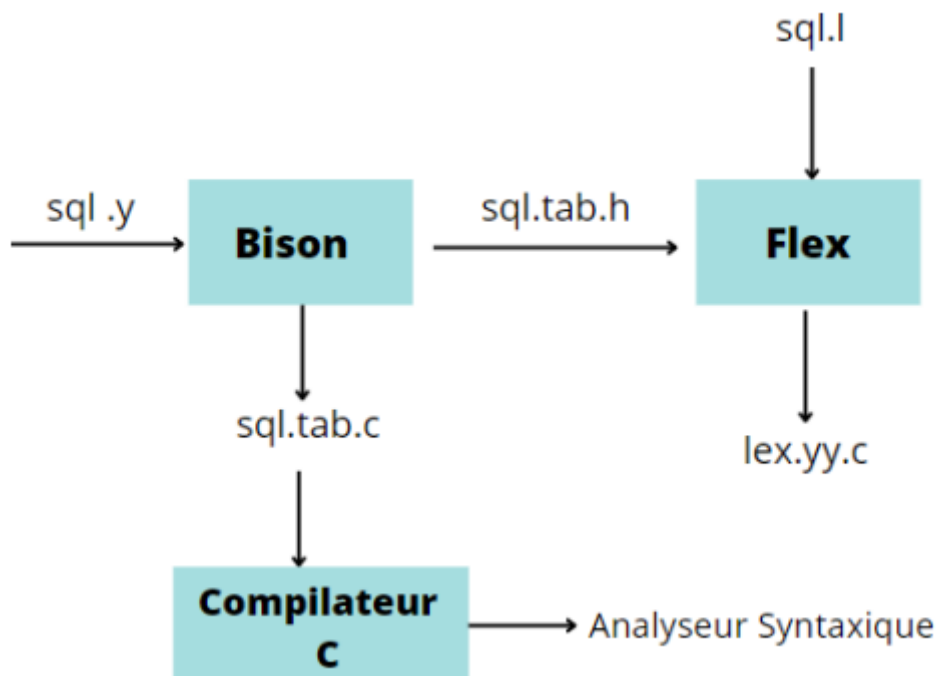
Flex est une version libre de l'analyseur lexical Lex. Il est en général associé à l'analyseur syntaxique GNU Bison, la version GNU de Yacc. C'est un outil de génération d'analyseurs syntaxiques, des programmes qui reconnaissent les motifs lexicaux dans le texte. Il accepte en entrée des unités lexicales sous formes d'expressions régulières et produit un programme écrit en langage C qui, une fois compilé, reconnaît ces unités lexicales. En sortie, flex génère un fichier source C, appelé 'lex.yy.c', qui définit une routine 'yylex()'. Ce fichier est compilé et lié avec l'option '-lfl' (correspondant à la bibliothèque flex) pour produire un programme exécutable. Lorsque l'exécutable est lancé, il scanne son entrée à la recherche d'occurrences d'expressions régulières précédentes. Pour chaque expression trouvée, il exécute le code C correspondant.

3.2 Bison

Bison (version GNU de YACC) est un outil de génération automatique d'analyseurs syntaxiques. Il accepte en entrée la description d'un langage sous forme d'une grammaire et produit un programme écrit en C qui, une fois compilé, reconnaît les mots ou les programmes appartenant au langage engendré par la grammaire en entrée. Un analyseur syntaxique écrit en Bison consiste en une description d'une GHC. Un programme Bison est un simple fichier texte enregistré avec l'extension ".y".



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



4 Réalisation

Notre projet consiste à développer un interpréteur acceptant les requêtes de manipulation et d'accès à une base de données. Les types de requêtes qui doivent être acceptés par l'interpréteur

- Le 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 :

```
%{  
#include <stdio.h>  
#include <conio.h>  
#include "sql.tab.h"  
int numID=0;  
int errlx=0;  
%}
```

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.

```
CREATE [Cc][Rr][Ee][Aa][Tt][Ee]  
TABLE [Tt][Aa][Bb][Ll][Ee]  
SELECT [Ss][Ee][Ll][Ee][Cc][Tt]  
DELETE [Dd][Ee][Ll][Ee][Tt][Ee]  
INSERT [Ii][Nn][Ss][Ee][Rr][Tt]  
INTO [Ii][Nn][Tt][Oo]  
FROM [Ff][Rr][Oo][Mm]  
WHERE [Ww][Hh][Ee][Rr][Ee]  
GROUP [Gg][Rr][Oo][Uu][Pp]  
ORDER [Oo][Rr][Dd][Ee][Rr]  
BY [Bb][Yy]  
ASC [Aa][Ss][Cc]  
DESC [Dd][Ee][Ss][Cc]  
STRING [Ss][Tt][Rr][Ii][Nn][Gg]  
NUMBER [Nn][Uu][Mm][Bb][Ee][Rr]  
VALUES [Vv][Aa][Ll][Uu][Ee][Ss]
```

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.

```

AND [Aa][Nn][Dd]
OR [Oo][Rr]
LESSorEQUAL ("<=")
GREATERorEQUAL (">=")
DISTINCT ("!=")
GREATER \>
LESSER \<
EQUAL \=
PAR_OUV \(
PAR_FER\)
PT_VIR \;
VIR \,
PT \.
COTE \'
DOUBLECOTE \"
ETOILE \*

```

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`.

File Edit Format View Help

```

%%
{PRINT}      {printf("\n Nombre des champs : %d ", numID); return 0;}
{IGNORE}     {}
{AND}        {printf(" AND %s\n", yytext); return AND;}
{OR}         {printf(" OR: %s\n", yytext); return OR;}
{PAR_OUV}    {printf(" PAR_OUV: %s\n", yytext); return PAR_OUV;}
{PAR_FER}    {printf(" PAR_FER: %s\n", yytext); return PAR_FER;}
{PT_VIR}     {printf(" PT_VIR: %s\n", yytext); return PT_VIR;}
{VIR}        {printf(" VIR: %s\n", yytext); ; return VIR;}
{PT}         {printf(" PT: %s\n", yytext); return PT;}
{COTE}       {printf(" COTE: %s\n", yytext); return COTE;}
{DOUBLECOTE} {printf(" DOUBLECOTE: %s\n", yytext); return DOUBLECOTE;}
{ETOILE}     {printf(" ETOILE: %s\n", yytext); return ETOILE;}
{GREATERorEQUAL} {printf(" GREATERorEQUAL: %s\n", yytext); return GREATERorEQUAL;}
{LESSorEQUAL} {printf(" LESSorEQUAL: %s\n", yytext);return LESSorEQUAL;}
{DISTINCT}   {printf(" DISTINCT: %s\n", yytext); return DISTINCT;}
{LESSER}     {printf(" LESSER: %s\n", yytext); return LESSER;}
{GREATER}    {printf(" GREATER: %s\n", yytext); return GREATER;}
{EQUAL}      {printf(" EQUAL: %s\n", yytext); return EQUAL;}
{CREATE}     {printf(" CREATE: %s\n", yytext); return CREATE;}
{TABLE}      {printf(" TABLE: %s\n", yytext); return TABLE;}
{SELECT}     {printf(" SELECT: %s\n", yytext); return SELECT;}
{DELETE}     {printf(" DELETE: %s\n", yytext); return DELETE;}
{INSERT}     {printf(" INSERT: %s\n", yytext); return INSERT;}
{INTO}       {printf(" INTO: %s\n", yytext); return INTO;}
{VALUES}     {printf(" VALUES: %s\n", yytext); return VALUES;}
{FROM}       {printf(" FROM: %s\n", yytext); return FROM;}
{WHERE}      {printf(" WHERE: %s\n", yytext); return WHERE;}
{GROUP}      {printf(" GROUP: %s\n", yytext); return GROUP;}
{ORDER}      {printf(" ORDER: %s\n", yytext); return ORDER;}
{BY}         {printf(" BY: %s\n", yytext); return BY;}
{ASC}        {printf(" ASC: %s\n", yytext); return ASC;}
{DESC}       {printf(" DESC: %s\n", yytext); return DESC;}
{NUMBER}     {printf(" NUMBER: %s\n", yytext) ; return NUMBER;}
{STRING}     {printf(" STRING: %s\n", yytext) ; return STRING;}
---
```

4.1.4 Variables à afficher :

Initialisation du nombre des champs et des erreurs.

```
int numID=0;  
int errlx=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 :

```
%{  
#include <math.h>  
#include <stdio.h>  
#include <conio.h>  
#include <stdlib.h>  
#include "lex.yy.c"  
int yylex();  
int yyerror();  
extern FILE *yyin;  
%}
```


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.

```
%token OR
%token LESSOrEQUAL
%token GREATEROrEQUAL
%token DISTINCT
%token GREATER
%token LESSER
%token EQUAL
%token CREATE
%token TABLE
%token SELECT
%token DELETE
%token INSERT
%token INTO
%token FROM
%token WHERE
%token GROUP
%token ORDER
%token BY
%token ASC
%token DESC
%token STRING
%token NUMBER
%token VALUES
%token PAR_OUV
%token PAR_FER
%token PT_VIR
%token VIR
%token PT
%token COTE
```

4.2.3 Définition de la grammaire :

Pour définir une grammaire qui commence avec la directive `%start grammaire`. Nous avons défini la grammaire, en commençant par l'axiome, qui est la règle de départ de la grammaire, suivie des règles de production pour chaque type de requête SQL.

L'axiome est défini comme une requête suivie d'un point-virgule et d'une boucle. La boucle permet la récursivité de la grammaire en autorisant des requêtes multiples.

Les requêtes possibles sont les suivantes :créer, sélectionner,delete, insertion . Chacune de ces requêtes est définie par une règle de production qui décrit la structure de la requête en termes de mots clés et de paramètres. Les autres règles de production décrivent les différents paramètres possibles pour chaque type de requête. Par exemple, la règle 'options' définit les options possibles pour une requête 'sélectionner', comme le tri des résultats ou le regroupement des données. Les règles 'opt1', 'opt2' et 'opt3' décrivent les différentes options de tri ou de regroupement possibles.

```

%start grammaire
%%
grammaire : Axiome;
Axiome : requete PT_VIR loop;
loop : Axiome | /*epsilon*/ ;
requete : creer | selectionner | delete | insertion;
creer : CREATE TABLE newTable;
newTable : ID PAR_OUV Colonne PAR_FER | ID ;
selectionner : SELECT id FROM table options;
options : opt1 | opt2 | opt3 | /*epsilon*/ ;
opt1 : WHERE exp opt4;
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 ;
insertion : INSERT INTO table VALUES PAR_OUV ident1 PAR_FER ;
ident1 : ident2 | COTE text COTE | NUM | DOUBLECOTE text DOUBLECOTE ;
ident2 : ID | ID PT ID;
id : identificatuers | ETOILE;
exp : ident1 operateur ident1 opcexp | PAR_OUV ident1 operateur ident1 opcexp PAR_FER opcexp;
opcexp : /*epsilon*/ | logique exp;
text : Ponct | Ponct text;
Ponct : ID | VIR | PT | NUM;
identificatuers : ident2 | ident2 VIR identificatuers;
Colonne : ID type | ID type VIR Colonne;
type : NUMBER | STRING;
table : ID | ID VIR table;
operateur : GREATERorEQUAL | LESSorEQUAL | GREATER | LESSER | DISTINCT | EQUAL;
logique : AND | OR;

%%

```

Afin d'afficher les messages d'erreurs, nous avons utilisé la fonction `int yyerror(char const *)`.

```

int yyerror(const char *str)
{
    if (errlx)
    {
        fprintf(stderr, "Erreur lexicale | Ligne: %d\n%s\n", yylineno, str);
    }
    else
    {
        fprintf(stderr, "Erreur syntaxique | Ligne: %d\n%s\n", yylineno, str);
    }
    getch();
}

```

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.

```
int main (int argc, char *argv[])
{
    printf("SQL requete : \n");
    if (argc == 2)
    {
        yyin = fopen (argv[1], "rt");
        if (yyin == NULL)
        {
            printf ("Impossible d'ouvrir le fichier %s\n", argv[1]);
            exit (-1);
        }
    }
    else
        yyin = stdin;
    yyparse();
    printf("");
    return 0;
};
```

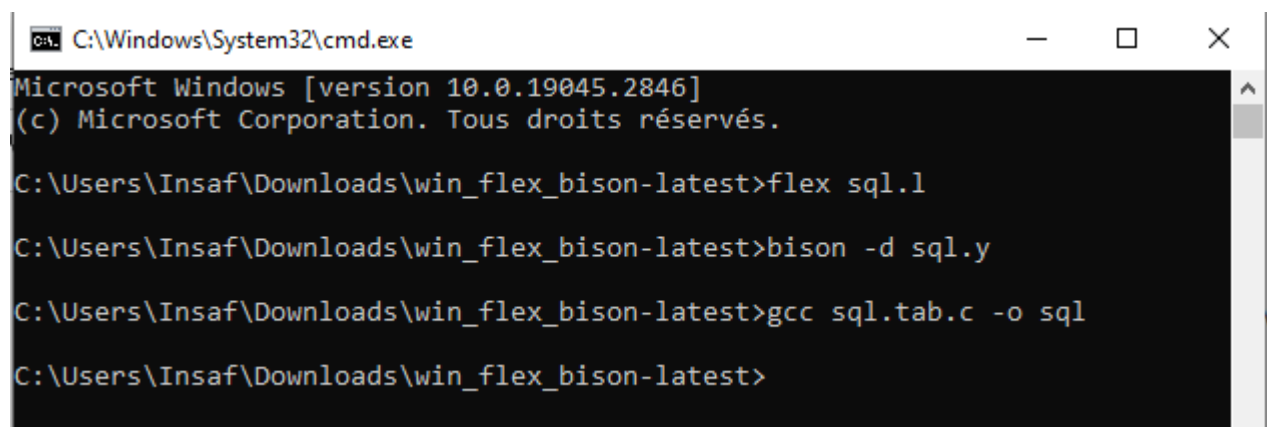
Pour compiler le fichier de spécification syntaxique bison nous avons commencé par taper la commande **bison -d sql.y**.

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 commençons par l'exécution des commandes suivantes :



```
C:\Windows\System32\cmd.exe
Microsoft Windows [version 10.0.19045.2846]
(c) Microsoft Corporation. Tous droits réservés.

C:\Users\Insaf\Downloads\win_flex_bison-latest>flex sql.l

C:\Users\Insaf\Downloads\win_flex_bison-latest>bison -d sql.y

C:\Users\Insaf\Downloads\win_flex_bison-latest>gcc sql.tab.c -o sql

C:\Users\Insaf\Downloads\win_flex_bison-latest>
```

Exécution à partir de l'invite de commande :

```
C:\Users\dhrrif\Downloads\win_flex_bison-latest(1)>sql.exe
SQL requete :
select * from personne;
SELECT: select
ETOILE: *
FROM: from
ID: personne
PT_VIR: ;
print
Nombre des champs : 1
C:\Users\dhrrif\Downloads\win_flex_bison-latest(1)>
```

Pour afficher une erreur dans une ligne au niveau de l'input, on tape les fausses requêtes sql suivantes : **selext num from personages;** et **select num fro pers;**

```
C:\Users\dhrrif\Downloads\win_flex_bison-latest(1)>sql.exe
SQL requete :
selext num from personages;
ID: selext
Erreur syntaxique | Ligne: 1
syntax error

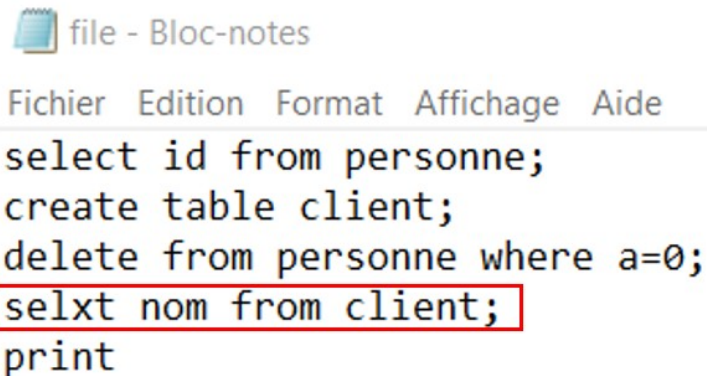
C:\Users\dhrrif\Downloads\win_flex_bison-latest(1)>sql.exe
SQL requete :
select num fro pers;
SELECT: select
ID: num
ID: fro
Erreur syntaxique | Ligne: 1
syntax error
```

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

```
C:\Users\Insaf\Downloads\win_flex_bison-latest>sql.exe
SQL requete :
select * from tab;
SELECT: select
ETOILE: *
FROM: from
ID: tab
PT_VIR: ;
create table;
ID: create
Erreur syntaxique | Ligne: 2
syntax error
```

D'après la capture ci-dessus nous pouvons constater que la première requête a été analysé par succès tandis que la deuxième ("create table") contient une erreur syntaxique. Le programme a détecté cette dernière et il a affiché comme sortie le numéro de la ligne de la requête (Ligne 2).

Notre programme permet aussi d'exécuter des requêtes SQL à partir d'un fichier. Pour ce faire, nous présentons ce fichier texte qui contient les requêtes suivantes :



```
file - Bloc-notes
Fichier Edition Format Affichage Aide
select id from personne;
create table client;
delete from personne where a=0;
selxt nom from client;
print
```

Nous avons introduit cette commande **sql.exe file.txt** pour exécuter le fichier contenant les requêtes. Nous constatons une erreur syntaxique au niveau du champ : selxt dans la ligne 4.

```
C:\Users\Insaf\Downloads\win_flex_bison-latest>sql.exe 1 file.txt
SQL requete :
SELECT: select
ID: id
FROM: from
ID: personne
PT_VIR: ;
CREATE: create
TABLE: table
ID: client
PT_VIR: ;
DELETE: delete
FROM: from
ID: personne
WHERE: where
ID: a
EQUAL: =
NUM: 0
PT_VIR: ;
ID: selxt
Erreur syntaxique | Ligne: 4
syntax error

C:\Users\Insaf\Downloads\win_flex_bison-latest>
```

6 Conclusion

Ce programme est un exemple d'implémentation d'un analyseur syntaxique pour le langage SQL en utilisant l'outil Flex pour la génération du lexer et en définissant les règles de production de la grammaire avec la syntaxe de Bison.

Le programme utilise également des bibliothèques standard telles que `math.h`, `stdio.h` et `stdlib.h` 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 entrées par l'utilisateur ou lues à partir d'un fichier.

En cas d'erreur, le programme affiche un message d'erreur avec le numéro de ligne correspondant.

