Higher Institute of Computer Science
University of Tunis El Manar

Département Génie Logiciel

MINI-PROJET COMPILATION

Interpréteur de Requêtes SQL avec Flex et Bison

Réalisé par:
Halbouni Nadhir
Bey Mehrez
Hajji Mohamed Amine

Encadrant Académique : **Wafa KAROUI**

Année académique 2023/2024

Sommaire

Introduction Générale

La compilation, au cœur du processus de développement logiciel, est un domaine crucial pour la transformation du code source en un programme exécutable. Ce processus complexe implique plusieurs étapes, dont l'analyse, la transformation et la génération de code, réalisées par différents outils et techniques. Parmi ces outils, Flex et Bison occupent une place importante. Flex, également connu sous le nom de Lex, est un générateur d'analyseurs lexicaux, tandis que Bison, ou Yacc, est un générateur d'analyseurs syntaxiques. Ensemble, ces outils permettent de créer des analyseurs de langage puissants et efficaces.

Objectif du projet :

Ce projet consiste à développer un interpréteur de requêtes de manipulation et d'accès à une base de données en utilisant les outils Flex et Bison. L'objectif est de concevoir un système capable d'accepter et de traiter des requêtes de type "create", "delete", "update" et "select", en exploitant les concepts de compilation, notamment l'analyse lexicale et syntaxique, pour créer un interpréteur précis et efficace.

C’est quoi un interpréteur SQL :

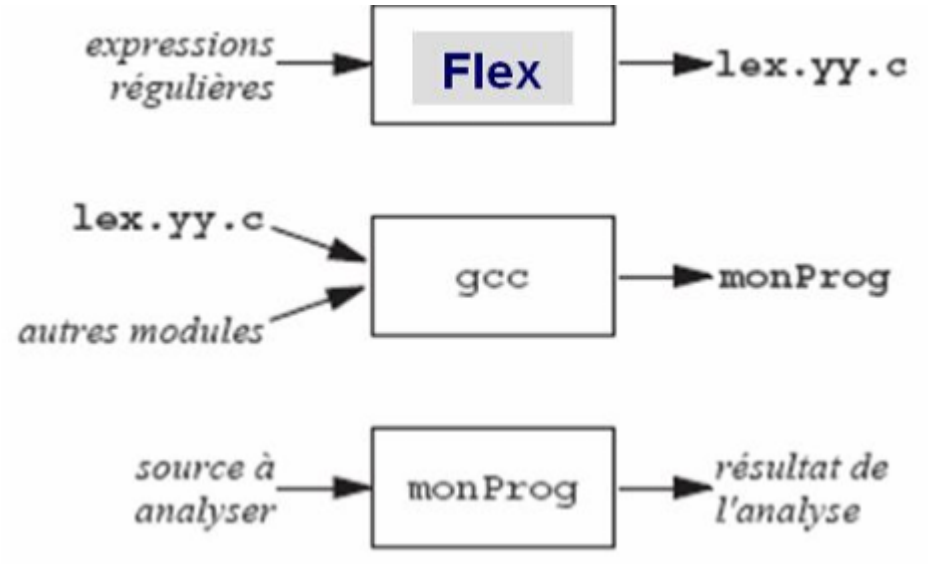
De manière générale, un interpréteur SQL est un programme informatique capable de comprendre et d'exécuter des requêtes écrites dans le langage SQL (Structured Query Language), permettant ainsi aux utilisateurs d'interagir avec une base de données de manière intuitive et efficace. Ce projet s'inscrit dans cette démarche en développant un interpréteur spécifique pour les requêtes SQL, afin de faciliter la manipulation et l'accès aux données dans un environnement de base de données.

Implementation :

L'Analyse Lexicale avec Flex :

L'analyseur lexical a été conçu avec Flex pour identifier et tokenizer les éléments clés des requêtes SQL. Flex a été choisi pour sa flexibilité et sa capacité à gérer des expressions régulières complexes, ce qui est essentiel pour analyser efficacement les requêtes SQL.

Conception de l'Analyseur Lexical:
L'analyseur lexical utilise des expressions régulières pour reconnaître et tokenizer les motifs spécifiques des requêtes SQL, tels que les mots-clés (SELECT, FROM, WHERE, etc.), les identifiants (noms de tables, colonnes, etc.) et les valeurs numériques. Chaque motif reconnu génère un token correspondant qui est ensuite utilisé par l'analyseur syntaxique Bison pour construire l'arbre syntaxique.
La figure suivante résume le processus de la réalisation d'un analyseur lexical en utilisant Flex :



Code :

- identificateurs réservés prédéfinis :

```
SELECT          { return SELECT; }
FROM            { return FROM; }
WHERE           { return WHERE; }
CREATE          { return CREATE; }
DELETE          { return DELETE; }
UPDATE          { return UPDATE; }
TABLE           { return TABLE; }
SET             { return SET; }
```

- opérateurs logiques :

```
AND             { yylval.chaine = strdup(yytext); return AND; }
OR              { yylval.chaine = strdup(yytext); return OR; }
"*"             { return ALL; }
```

- types des données et opérateurs de contrainte :

```
"INT"|"VARCHAR("([0-9]+)")"|"BOOL" { return DATATYPE; }
"NOT NULL"|"UNIQUE"|"PRIMARY KEY" { return OPTIONS; }
```

- TRUE et FALSE :

```
"TRUE"|"FALSE" { return BOOLEAN; }
```

- Identificateurs et nombres :

```
[a-zA-Z_][a-zA-Z0-9_]* { yylval.chaine = strdup(yytext); return IDENTIFIER; }
[0-9]+          { yylval.chaine = strdup(yytext); return NUMBER; }
```

- opérateur égal, virgules et parenthèses :

```
"="             { yylval.chaine = strdup(yytext); return EQ; }
","             { yylval.chaine = strdup(yytext); return COMMA; }
";"             { yylval.chaine = strdup(yytext); return SEMICOLON; }
"("             { yylval.chaine = strdup(yytext); return OPENPAR; }
")"             { yylval.chaine = strdup(yytext); return CLOSEPAR; }
```

- opérateurs relationnels :

```
"<="            { yylval.chaine = strdup(yytext); return OPREL; }
">="            { yylval.chaine = strdup(yytext); return OPREL; }
"<"             { yylval.chaine = strdup(yytext); return OPREL; }
">"             { yylval.chaine = strdup(yytext); return OPREL; }
"!="            { yylval.chaine = strdup(yytext); return DIFF; }
```

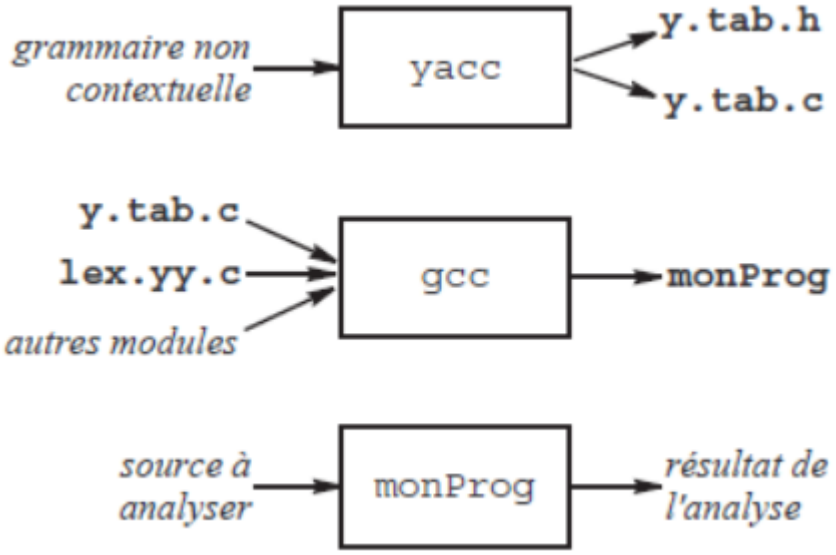
- autres caractères non reconnus :

```
.               { fprintf(stderr, "Error: Unrecognized character '%s'\n", yytext); }
```

L'Analyse Syntaxique avec Bison:

L'analyse syntaxique a été mise en œuvre en utilisant Bison pour définir la grammaire des requêtes SQL. Bison a été choisi pour sa puissance et sa flexibilité pour définir des règles de grammaire complexes et construire un arbre syntaxique correspondant.

Conception de l'Analyseur Syntaxique :
La grammaire a été conçue pour prendre en charge les requêtes CREATE, DELETE, UPDATE et SELECT avec leurs clauses et conditions respectives. Des règles spécifiques ont été définies pour chaque type de requête et chaque élément de la requête, tels que les clauses SELECT, FROM, WHERE, les identifiants de tables et de colonnes, et les valeurs numériques.
Le programme yacc est un tel générateur d'analyseurs syntaxiques. Il prend en entrée un fichier source constitué essentiellement des productions d'une grammaire non contextuelle G, et sort à titre de résultat un programme C qui, une fois compilé, est un analyseur syntaxique pour le langage L(G).



CODE :

- axiome :

```
%start query
```

- tokens:

```
%token SELECT FROM WHERE CREATE DELETE UPDATE ALL COMMA SEMICOLON TABLE DATATYPE OPENPAR CLOSEPAR OPTIONS SET BOOLEAN
%token <chaine> AND
%token <chaine> OR
%token <chaine> DIFF
%token <chaine> OPREL
%token <chaine> EQ
%token <chaine> NUMBER
```

```
%token <chaine> IDENTIFIER
```

- Non terminaux (de type chaîne pour l'analyse sémantique) :

```
%type <chaine> condition
%type <chaine> identifierOrNumber
%type <chaine> table_fields
%type <chaine> operator
```

- règles de production :

```
query: select_query | create_query | delete_query | update_query
     | select_query query | create_query query | delete_query query | update_query query

select_query: SELECT ALL FROM IDENTIFIER SEMICOLON
            | SELECT ALL FROM IDENTIFIER WHERE condition SEMICOLON
            | SELECT table_fields FROM IDENTIFIER SEMICOLON
            | SELECT table_fields FROM IDENTIFIER WHERE condition SEMICOLON

create_query: CREATE TABLE IDENTIFIER OPENPAR fields CLOSEPAR SEMICOLON

delete_query: DELETE FROM IDENTIFIER  SEMICOLON
            | DELETE FROM IDENTIFIER WHERE condition SEMICOLON

update_query: UPDATE IDENTIFIER SET update_fields SEMICOLON

update_fields: IDENTIFIER EQ NUMBER
             | IDENTIFIER EQ BOOLEAN
             | update_fields COMMA update_fields

fields: IDENTIFIER DATATYPE OPTIONS | fields COMMA fields | IDENTIFIER DATATYPE

table_fields: IDENTIFIER
            | IDENTIFIER COMMA table_fields

condition: IDENTIFIER operator identifierOrNumber
         | condition AND condition
         | condition OR condition

operator: DIFF  | OPREL | EQ
identifierOrNumber: IDENTIFIER | NUMBER
```

- message d'erreur en cas d'une requête invalide:

```
void yyerror(const char *s) {
    fprintf(stderr, "Error: Unknown SQL Query !!\n");
}
```

Ajout d'Actions Sémantiques :

Des actions sémantiques ont été ajoutées pour effectuer des calculs et détecter les erreurs lors de l'analyse des requêtes. Les actions sémantiques sont utilisées pour

effectuer des opérations spécifiques, telles que le calcul du nombre de champs à sélectionner dans une requête SELECT, la vérification de la validité des identifiants et des valeurs, et la gestion des erreurs de syntaxe et de sémantique.

CODE :

- entier cpt pour tenir compte du nombre des colonnes sélectionnées :

```
int cpt = 0;
```

- union pour différencier entre les types de tokens (entier, chaîne, reel, ...) :

```
%union {
    int entier;
    double reel;
    char* chaine;
}
```

- ajout de la sémantique au règles de production :

```
query: select_query | create_query | delete_query | update_query
    | select_query query | create_query query | delete_query query | update_query query

select_query: SELECT ALL FROM IDENTIFIER SEMICOLON
              { printf("all elements from table %s are selected\n", $4); }
            | SELECT ALL FROM IDENTIFIER WHERE condition SEMICOLON
              { printf("all elements from table %s respecting this condition %s are selected\n", $4, $6); }
            | SELECT table_fields FROM IDENTIFIER SEMICOLON
              { printf("fields %s are selected from table %s\n", $2, $4);  printf("And you selected %d columns\n", cpt); cpt = 0;}
            | SELECT table_fields FROM IDENTIFIER WHERE condition SEMICOLON
              { printf("fields %s from table %s which respect this condition %s are selected\n", $2, $4, $6);
                printf("And you selected %d columns\n", cpt); cpt = 0;}

create_query: CREATE TABLE IDENTIFIER OPENPAR fields CLOSEPAR SEMICOLON { printf("table %s created successfully !\n", $3); }

delete_query: DELETE FROM IDENTIFIER  SEMICOLON
              { printf("all elements from table %s are deleted\n", $3); }
            | DELETE FROM IDENTIFIER WHERE condition SEMICOLON
              { printf("all elements from table %s respecting this condition %s are deleted\n", $3, $5); }

update_query: UPDATE IDENTIFIER SET update_fields SEMICOLON
              { printf("table %s updated successfully\nAnd you updated %d columns\n", $2, cpt); cpt = 0;}

update_fields: IDENTIFIER EQ NUMBER { cpt++; }
             | IDENTIFIER EQ BOOLEAN  { cpt++; }
             | update_fields COMMA update_fields

fields: IDENTIFIER DATATYPE OPTIONS | fields COMMA fields | IDENTIFIER DATATYPE

table_fields: IDENTIFIER { $$ = strdup($1); cpt++;}
            | IDENTIFIER COMMA table_fields { $$ = strcat(strcat(strdup($1), ", "), $3); cpt++;}

condition: IDENTIFIER operator identifierOrNumber { $$ = strcat(strcat($1, $2), $3); }
         | condition AND condition { $$ = strcat(strcat($1, " AND "), $3); }
         | condition OR condition { $$ = strcat(strcat($1, " OR "), $3); }

operator: DIFF { $$ = strdup($1); } | OPREL { $$ = strdup($1); } | EQ { $$ = strdup($1); }
identifierOrNumber: IDENTIFIER { $$ = strdup($1); } | NUMBER { $$ = strdup($1); }
```

Résultats :

L'interpréteur SQL a été testé avec une série de requêtes SQL pour valider son fonctionnement et identifier les éventuels problèmes ou erreurs. Les tests ont couvert différentes fonctionnalités et cas d'utilisation pour garantir la robustesse et la

fiabilité de l'interpréteur. Des tests ont été réalisés avec différents types de requêtes SQL, y compris des requêtes valides et des requêtes incorrectes pour évaluer la capacité de l'interpréteur à détecter et signaler les erreurs.

- *Cas de Test 1 : SELECT :*

```
SELECT * FROM USERS;
all elements from table USERS are selected
```

```
SELECT * FROM USERS WHERE age > 30 AND height = 170 OR weight <= 60;
all elements from table USERS respecting this condition age>30 AND height=170 OR weight<=60 are selected
```

```
SELECT user_id, username, age FROM USERS;
fields user_id, username, age are selected from table USERS
And you selected 3 columns
```

```
SELECT user_id, age FROM USERS WHERE age < 30 AND height = 170 OR weight > 70;
fields user_id, age from table USERS which respect this condition age<30 AND height=170 OR weight>70 are selected
And you selected 2 columns
```

- *Cas de Test 2 : CREATE :*

```
CREATE TABLE USERS(user_id INT PRIMARY KEY, username VARCHAR(20) NOT NULL, age INT);
table USERS created successfully !
```

- *Cas de Test 3 : UPDATE :*

```
UPDATE USERS SET age = 34, married = FALSE, height = 173;
table USERS updated successfully
And you updated 3 columns
```

- *Cas de Test 4 : DELETE :*

```
DELETE FROM USERS;
all elements from table USERS are deleted
```

```
DELETE FROM USERS WHERE age > 50 AND height < 180 OR weight != 60;
all elements from table USERS respecting this condition age>50 AND height<180 OR weight!=60 are deleted
```

- Cas de Test 5 : Wrong Query :

```
SELECT FROM ALL USERS;
Error: Unknown SQL Query !!
```

Conclusion:

Ce projet a permis de développer un interpréteur SQL simple mais fonctionnel en utilisant les outils Flex et Bison. L'analyseur lexical et syntaxique conçu est capable d'analyser et de traiter une variété de requêtes SQL, tout en détectant et signalant les erreurs éventuelles. Le projet a été une expérience enrichissante pour comprendre le processus de conception et de développement d'un interpréteur de langage de requête. Des améliorations et des extensions peuvent être apportées à l'interpréteur pour prendre en charge davantage de fonctionnalités SQL et optimiser les performances à l'avenir.