
Technique de compilation

Interpréteur avec flex/bison

Joumene Ben Said

Plan

1. Objectif
2. Analyse lexicals et analyse syntaxique
3. Outils utilisés
4. Table de lexèmes
5. Grammaires
6. Réalisation



Objectif

L'objectif de ce mini-projet est de développer un interpréteur pour les requêtes de manipulation et d'accès à une base de données en utilisant les outils Flex/Bison. L'interpréteur doit être capable d'accepter les requêtes de type create/delete/update et select, et d'exécuter des requêtes sur une base de données. Il doit également être capable de détecter et signaler à l'utilisateur différents types d'erreurs qui peuvent se produire lors de l'exécution de ces requêtes.

Analyse lexicales et analyse syntaxique

Analyse lexicale :

L'analyseur lexical est responsable de la reconnaissance des différents symboles qui apparaissent dans la requête. Voici la spécification de l'analyseur lexical pour notre interpréteur :

1. Les mots clés de la requête (create, delete, update, select) sont reconnus et renvoyés sous forme de tokens appropriés.
2. Les noms de tables, les noms de colonnes, les valeurs et les opérateurs sont également reconnus et renvoyés sous forme de tokens.
3. Les espaces et les commentaires sont ignorés.

Analyse syntaxique :

L'analyseur syntaxique est responsable de la vérification de la structure de la requête. Voici la spécification Bison pour notre interpréteur :

1. La grammaire doit permettre de reconnaître les différentes requêtes (create, delete, update, select) ainsi que les clauses associées (where, from, set, etc.).
2. La grammaire doit permettre de construire l'arbre syntaxique correspondant à la requête entrée par l'utilisateur.

Outils utilisés

Bison & Flex :

Bison et Flex (anciennement appelé Lex) sont deux outils logiciels utilisés pour construire un compilateur ou un interprète pour un langage de programmation.



Flex est un générateur de lexers (analyseurs lexicaux), qui permet de convertir une séquence de caractères en une séquence de "jetons" (tokens) significatifs dans le langage de programmation. Il est utilisé pour automatiser la tâche fastidieuse de l'analyse lexicale.

Bison, quant à lui, est un générateur de parseurs qui prend une grammaire contextuelle libre comme entrée et génère un analyseur syntaxique en C ou C++ qui peut parser une entrée conformément à cette grammaire. Il utilise les tokens produits par le lexer pour générer un arbre syntaxique qui peut être utilisé pour l'analyse sémantique et la génération de code.

Ensemble, Flex et Bison peuvent être utilisés pour créer un pipeline de compilation complet, de la saisie du code source jusqu'à la génération de code exécutable.



Make:

Make est un outil de construction (ou build tool) utilisé pour automatiser le processus de compilation de programmes. Il permet de décrire les dépendances entre différents fichiers et les commandes nécessaires pour construire un programme à partir de ces fichiers. En utilisant un fichier Makefile, l'utilisateur peut décrire les règles qui permettent de construire le programme en spécifiant les dépendances entre les fichiers sources, les fichiers objets et les exécutables, ainsi que les commandes nécessaires pour construire chaque étape du processus.

Make utilise le concept de timestamp pour déterminer si un fichier doit être recompilé. Il compare la date de modification des fichiers sources et des fichiers objets pour déterminer s'ils sont à jour. Si un fichier source a été modifié depuis la dernière compilation, Make exécute les commandes nécessaires pour recompiler les fichiers objets correspondants.

Make est largement utilisé dans le développement de logiciels pour automatiser le processus de construction de programmes, notamment dans les projets qui contiennent de nombreux fichiers sources et des dépendances complexes.

Table de lexèmes

Lexème	Token
DELETE	DELETE
UPDATE	UPDATE
CREATE	CREATE
TABLE	TABLE
FROM	FROM
WHERE	WHERE
PRIMARY KEY	PRIMARY_KEY
VARCHAR	DATATYPE
DATE	DATATYPE
INT	DATATYPE

AND

AND

NOT

NOT

OR

OR

=

EQ

!=

NE

<

LT

<=

LE

>

GT

>=

GE

LIKE

LIKE

IN


IN

BETWEEN

BETWEEN

,

COMMA



(LPAREN
)	RPAREN
[0-9] +	NUMBER
"([^\n]	(\.)) * "
'([^\n]	(\.)) * '
[A-Za-z_][A-Za-z0-9_]*	IDENTIFIER

Grammaires

En compilation, une grammaire est une représentation formelle de la syntaxe d'un langage de programmation. Cette grammaire permet de décrire les règles qui gouvernent la structure des programmes dans le langage en question. Plus précisément, une grammaire de compilation décrit les règles syntaxiques d'un langage de programmation, c'est-à-dire la façon dont les différents éléments du programme peuvent être combinés pour former des constructions valides dans le langage. La grammaire est souvent définie sous forme de règles de production, qui spécifient comment les éléments du langage peuvent être combinés pour former des constructions plus complexes.

-Pour notre projet, on a utilisé les grammaires suivantes:

Grammaire pour les requêtes possibles:

```
query_statement: delete_statement
                | create_statement
                | select_statement
                | update_statement
                ;
```

Grammaire pour la requête update:

```
update_statement: UPDATE IDENTIFIER SET operation_list WHERE condition_list { printf("update query valid\n"); }
operation_list: IDENTIFIER operation value
               | IDENTIFIER operation value COMMA operation_list
operation: EQ
          | STAR
```

```

select_statement: SELECT select_list FROM table_list where_clause groupby_clause orderby_clause limit_clause {
    printf("select request valid.");
    selected_field_count = 0;
}

where_clause: WHERE condition_list
;

groupby_clause: GROUP_BY identifier_list
;

orderby_clause: ORDER_BY identifier_list order_direction
;

order_direction: ASC | DESC
;

limit_clause: LIMIT NUMBER
;

```

Grammaire pour la requête select:

```

select_list_item: IDENTIFIER
{
    selected_field_count++;
}
| IDENTIFIER '.' IDENTIFIER
{
    selected_field_count++;
}
| function_call AS IDENTIFIER
;

select_list: STAR
| select_list_item
| select_list COMMA select_list_item
;

identifier_list: IDENTIFIER
| STAR
| function_call
| IDENTIFIER COMMA identifier_list
;

```

```

function_call: AVG LPAREN args RPAREN
              | COUNT LPAREN args RPAREN
              | SUM LPAREN args RPAREN
              | FIRST LPAREN args RPAREN
              | LAST LPAREN args RPAREN
              | MIN LPAREN args RPAREN
              | MAX LPAREN args RPAREN
              ;

args: args COMMA NUMBER
     | NUMBER
     | STAR
     | IDENTIFIER
     ;

```

Grammaire pour la requête delete:

```

delete_statement: DELETE delete_list FROM table_list WHERE condition_list { printf("delete query valid\n"); }
                 ;

delete_list: IDENTIFIER
            | IDENTIFIER COMMA delete_list

condition_list: condition { }
              | condition_list OR condition { }
              | condition_list AND condition { }
              ;

condition: IDENTIFIER op value { }
          | IDENTIFIER BETWEEN value AND value
          | LPAREN condition RPAREN { }
          | NOT condition { }
          ;

op: EQ { }
   | NE { }
   | LT { }
   | LE { }
   | GT { }
   | GE { }
   | LIKE { }
   | IN { }
   ;

```

```

value: STRING_VALUE { }
      | NUMBER { }
      | IDENTIFIER { }
      | LPAREN value_list RPAREN { }
      ;

value_list: value { }
          | value COMMA value_list { }
          ;

table_list: IDENTIFIER
           | IDENTIFIER IDENTIFIER
           | IDENTIFIER COMMA table_list
           | IDENTIFIER IDENTIFIER COMMA table_list

```

Grammaire pour la requête create:

```

create_statement: CREATE TABLE IDENTIFIER LPAREN column_def_list RPAREN { printf("create query valid\n"); }
                ;

column_def_list: column_def { }
               | column_def_list COMMA column_def { }
               ;

column_def: IDENTIFIER data_type { }
           | IDENTIFIER data_type PRIMARY_KEY { }
           | IDENTIFIER data_type REFERENCES IDENTIFIER LPAREN IDENTIFIER RPAREN { }
           | primary_key_constraint { }
           | unique_constraint { }
           | foreign_key_constraint { }
           ;

primary_key_constraint: PRIMARY_KEY { }
                     ;

unique_constraint: UNIQUE { }
                 | UNIQUE LPAREN IDENTIFIER RPAREN { }
                 ;

foreign_key_constraint: FOREIGN_KEY LPAREN IDENTIFIER RPAREN REFERENCES IDENTIFIER LPAREN IDENTIFIER RPAREN { }
                     ;

data_type: DATATYPE { }
         | DATATYPE LPAREN NUMBER RPAREN { }
         | DECIMAL LPAREN NUMBER COMMA NUMBER RPAREN { }

```

Réalisation

Makefile:

Pour compiler notre projet, on doit utiliser la commande: **make**

```
M makefile
1  all:
2      flex -o sql_lexer.c --header=sql_lexer.h sql_lexer.l
3      bison -d sql_parser.y
4      gcc sql_parser.tab.c sql_lexer.c -o sql_interpreter.exe
5      ./sql_interpreter.exe input.txt
6
7  clean:
8      rm *.o *.c *.h *.exe
```

Exemples requêtes create:

```
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```



```

./sql_interpreter.exe input.txt
CREATE ----> *** CREATE Lexeme ***
TABLE ----> *** TABLE Lexeme ***
orders ----> *** IDENTIFIER Lexeme ***
( ----> *** LPAREN Lexeme ***
, ----> *** COMMA Lexeme ***
order_date ----> *** IDENTIFIER Lexeme ***
DATE ----> *** DATATYPE Lexeme ***
, ----> *** COMMA Lexeme ***
FOREIGN KEY ----> *** FOREIGN KEY Lexeme ***
( ----> *** LPAREN Lexeme ***
customer_id ----> *** IDENTIFIER Lexeme ***
) ----> *** RPAREN Lexeme ***
REFERENCES ----> *** REFERENCES Lexeme ***
customers ----> *** IDENTIFIER Lexeme ***
( ----> *** LPAREN Lexeme ***
customer_id ----> *** IDENTIFIER Lexeme ***
) ----> *** RPAREN Lexeme ***
) ----> *** RPAREN Lexeme ***
create query valid
; ;

```

Exemples requêtes delete:

```
DELETE b, v FROM birthdays b WHERE b = '1976-09-28';
```

```

./sql_interpreter.exe input.txt
DELETE ----> *** DELETE Lexeme ***
b ----> *** IDENTIFIER Lexeme ***
, ----> *** COMMA Lexeme ***
v ----> *** IDENTIFIER Lexeme ***
FROM ----> *** FROM Lexeme ***
birthdays ----> *** IDENTIFIER Lexeme ***
b ----> *** IDENTIFIER Lexeme ***
WHERE ----> *** WHERE Lexeme ***
b ----> *** IDENTIFIER Lexeme ***
= ----> *** EQ Lexeme ***
'1976-09-28' ----> *** STRING Lexeme ***
; ;
delete query valid

```

Exemples requêtes select:

```
select * from enseignant where class = 'IDISC1' AND NOM LIKE 'R*'
```

```
./sql_interpreter.exe input.txt
select ----> *** SELECT Lexeme ***
* ----> *** STAR Lexeme ***
from ----> *** FROM Lexeme ***
enseignant ----> *** IDENTIFIER Lexeme ***
where ----> *** WHERE Lexeme ***
class ----> *** IDENTIFIER Lexeme ***
= ----> *** EQ Lexeme ***
'IDISC1' ----> *** STRING Lexeme ***
AND ----> *** AND Lexeme ***
NOM ----> *** IDENTIFIER Lexeme ***
LIKE ----> *** LIKE Lexeme ***
'R*' ----> *** STRING Lexeme ***
; ;
```

Exemples requêtes update:

```
UPDATE employees SET salary = 60000 WHERE employee_id = 1234;
```

```
./sql_interpreter.exe input.txt
UPDATE ----> *** UPDATE Lexeme ***
employees ----> *** IDENTIFIER Lexeme ***
SET ----> *** SET Lexeme ***
salary ----> *** IDENTIFIER Lexeme ***
= ----> *** EQ Lexeme ***
60000 ----> *** NUMBER Lexeme ***
WHERE ----> *** WHERE Lexeme ***
employee_id ----> *** IDENTIFIER Lexeme ***
= ----> *** EQ Lexeme ***
1234 ----> *** NUMBER Lexeme ***
; ;
update query valid
```