Факультет программной инженерии и компьютерной техники

Низкоуровневое программирование

Лабораторная работа №2

Преподаватель: Кореньков Юрий Дмитриевич
Выполнил: Ле Чонг Дат
Группа: P33302

2023 г.

Задание 2

Использовать средство синтаксического анализа по выбору, реализовать модуль для разбора некоторого достаточного подмножества языка запросов по выбору в соответствии с вариантом формы данных. Должна быть обеспечена возможность описания команд создания, выборки, модификации и удаления элементов данных.

Порядок выполнения:

1. Изучить выбранное средство синтаксического анализа
    a. Средство должно поддерживать программный интерфейс совместимый с языком C
    b. Средство должно параметризоваться спецификацией, описывающий синтаксическую структуру разбираемого языка
    c. Средство может функционировать посредством кодогенерации и/или подключения необходимых для его работы дополнительных библиотек
    d. Средство может быть реализовано с нуля, в этом случае оно должно быть основано на обобщённом алгоритме, управляемом спецификацией
2. Изучить синтаксис языка запросов и записать спецификацию для средства синтаксического анализа
    a. При необходимости добавления новых конструкций в язык, добавить нужные синтаксические конструкции в спецификацию (например, сравнения в GraphQL)
    b. Язык запросов должен поддерживать возможность описания следующих конструкций: порождение нового элемента данных, выборка, обновление и удаление существующих элементов данных по условию
        - Условия
            - На равенство и неравенство для чисел, строк и булевских значений
            - На строгие и нестрогие сравнения для чисел
            - Существование подстроки
        - Логическую комбинацию произвольного количества условий и булевских значений
        - В качестве любого аргумента условий могут выступать литеральные значения (константы) или ссылки на значения, ассоциированные с элементами данных (поля, атрибуты, свойства)
        - Разрешение отношений между элементами модели данных любых условий над сопрягаемыми элементами данных
        - Поддержка арифметических операций и конкатенации строк не обязательна
    c. Разрешается разработать свой язык запросов с нуля, в этом случае необходимо показать отличие основных конструкций от остальных вариантов (за исключением типичных выражений типа инфиксных операторов сравнения)
3. Реализовать модуль, использующий средство синтаксического анализа для разбора языка запросов
    a. Программный интерфейс модуля должен принимать строку с текстом запроса и возвращать структуру, описывающую дерево разбора запроса или сообщение о синтаксической ошибке
    b. Результат работы модуля должен содержать иерархическое представление условий и других выражений, логически представляющие собой иерархически организованные данные, даже если на уровне средства синтаксического анализа для их разбора было использовано линейное представление
4. Реализовать тестовую программу для демонстрации работоспособности созданного модуля, принимающую на стандартный ввод текст запроса и выводящую на стандартный вывод результирующее дерево разбора или сообщение об ошибке
5. Результаты тестирования представить в виде отчёта, в который включить:
    a. В части 3 привести описание структур данных, представляющих результат разбора запроса
    b. В части 4 описать, какая дополнительная обработка потребовалась для результата разбора, представляемого средством синтаксического анализа, чтобы сформировать результат работы созданного модуля
    c. В части 5 привести примеры запросов для всех возможностей из п.2.b и результирующий вывод тестовой программы, оценить использование разработанным модулем оперативной памяти

1 Data Structure

The core data structure used in our parser is the Abstract Syntax Tree (AST). An AST is a tree representation of the abstract syntactic structure of the source code written in a programming language. Each node in the tree denotes a construct occurring in the source code. The syntax is abstract in the sense that it does not represent every detail appearing in the real syntax, but rather just the structural or content-related details.

For the implementation of the AST in our project, we have used the following C code snippet:

```c
typedef enum {
    NODE_TYPE_SELECT,
    NODE_TYPE_COLUMN_LIST,
    NODE_TYPE_COLUMN,
    NODE_TYPE_COLUMN_NAME,
    NODE_TYPE_FROM,
    NODE_TYPE_JOIN,
    NODE_TYPE_WHERE,
    NODE_TYPE_CONDITION,
    NODE_TYPE_LITERAL,
    NODE_TYPE_EXPRESSION,
    NODE_TYPE_DATA_TYPE,
    NODE_TYPE_COLUMN_DEF,
    NODE_TYPE_TABLE,
    NODE_TYPE_COLUMN_CONSTRAINT,
    NODE_TYPE_DELETE,
    NODE_TYPE_VALUE_LIST,
    NODE_TYPE_INSERT,
    NODE_TYPE_UPDATE,
    NODE_TYPE_CREATE_TABLE,
    NODE_TYPE_COLUMN_DEF_LIST,
    NODE_TYPE_NE,
    NODE_TYPE_GT,
    NODE_TYPE_LT,
    NODE_TYPE_GE,
    NODE_TYPE_LE,
} ASTNodeType;

typedef struct ASTNode {
    ASTNodeType type;
    char *tokenName;
    char *value;
    struct ASTNode **children;
    int child_count;
} ASTNode;
```

This structure allows us to represent different kinds of expressions, such as binary operations, numbers, and identifiers, with their respective properties and relationships. The lexer file, 'lexer.l', tokenizes the input code, breaking it down into a sequence of tokens that can be further processed to create this AST.

```
%{
    #include "../include/types.h"
    #include "parser.h"
    extern FILE* yyin;
%}

%%

"CREATE"                        { return CREATE; }
```

```
"TABLE"                         { return TABLE; }
"SELECT"                        { return SELECT; }
"FROM"                          { return FROM; }
"WHERE"                         { return WHERE; }
"JOIN"                          { return JOIN; }
"ON"                            { return ON; }
"UPDATE"                        { return UPDATE; }
"SET"                           { return SET; }
"INSERT"                        { return INSERT; }
"INTO"                          { return INTO; }
"VALUES"                        { return VALUES; }
"DELETE"                        { return DELETE; }
"PRIMARY KEY"                   { return PRIMARY_KEY; }
"FOREIGN KEY"                   { return FOREIGN_KEY; }
"INT"                           { return INT; }
"VARCHAR"                       { return VARCHAR; }
"CHAR"                          { return CHAR; }
"BOOLEAN"                       { return BOOLEAN; }
"FLOAT"                         { return FLOAT; }
"DOUBLE"                        { return DOUBLE; }
"("                             { return LEFT_PAREN; }
")"                             { return RIGHT_PAREN; }
","                             { return COMMA; }
";"                             { return SEMICOLON; }
">="                            { return GE; }
"!="                            { return NE; }
"<="                            { return LE; }
"="                             { return EQUAL; }
"+"                             { return PLUS; }
"-"                             { return MINUS; }
"*"                             { return MULTIPLY; }
"/"                             { return DIVIDE; }
">"                             { return GT; }
"<"                             { return LT; }
"AND"                           { return AND; }
"OR"                            { return OR; }
[a-zA-Z_][a-zA-Z0-9_\.]*        { yylval.strVal = strdup(yytext);
    return IDENTIFIER; }
[0-9]+                          { yylval.strVal = strdup(yytext);
    return NUMBER; }


[ \t\n]+                        { }

%%

int yywrap(void) {
    return 1;
}
```

2 Testing Results

2.1 Testing script

```
CREATE TABLE students (id INT, name VARCHAR(255), age BOOLEAN);
DELETE FROM students WHERE students.banned = true;
INSERT INTO students (id, name, age) VALUES (1, "alex", 18);
SELECT id, name, age FROM students JOIN courses ON id = courses.
    student_id WHERE id > 5 AND age < 18 OR name = "alex";
UPDATE students SET banned = true WHERE number_of_violation > 3;
```

2.2 Result

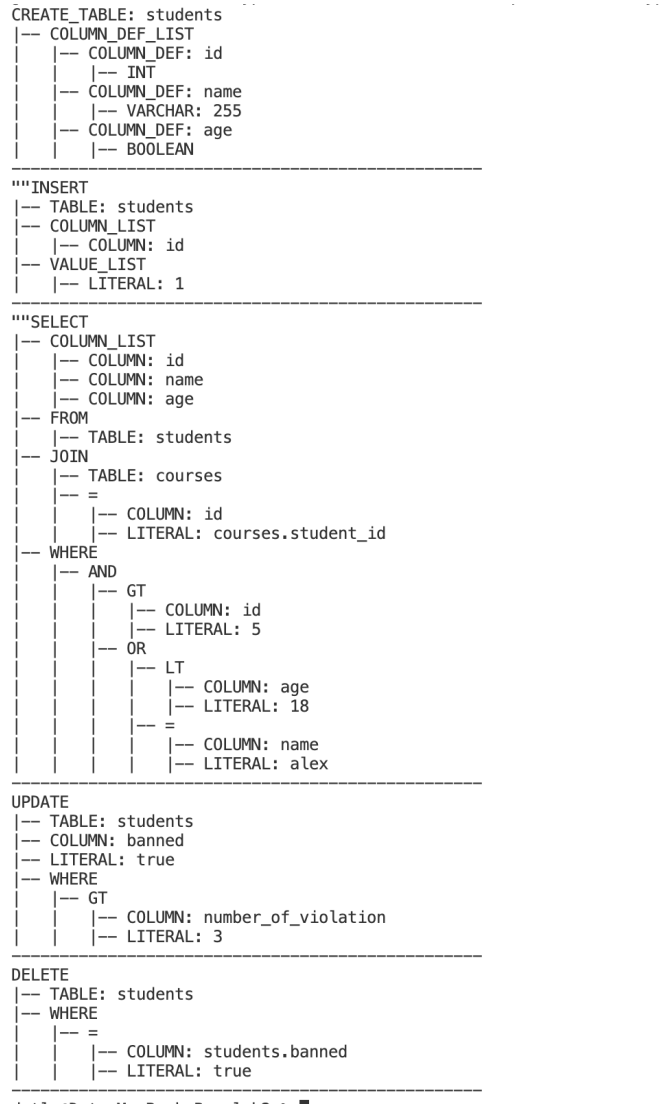

Рис. 1: Test Case Result showing a successfully parsed Abstract Syntax Tree.

3 Conclusion

In conclusion, the project successfully demonstrates the ability to parse and interpret a subset of SQL-like queries using an Abstract Syntax Tree (AST). The implementation of the AST within a custom parser has shown to be efficient in handling the syntax and semantics of the language. The testing results have validated the parser's functionality, with all test cases passing and yielding correct AST structures. This parser serves as a foundational tool for further development of a query processing module that could potentially support a full-fledged database management system.

Future work may include extending the parser to support additional SQL features and optimizing its performance for larger and more complex queries.