**Загальний опис проекту: Синтаксичний аналізатор арифметичних виразів на C++ із використанням Flex і Bison**

Цей проект — синтаксичний аналізатор для арифметичних виразів, розроблений на мові C++ із використанням Flex та Bison для лексичного та синтаксичного аналізу. Програма отримує на вхід арифметичний вираз, будує абстрактне синтаксичне дерево (AST) та обчислює значення виразу.

Мета проекту

Метою проекту є створення інструменту для розбору та обчислення арифметичних виразів, використовуючи інструменти Flex і Bison. В ході розробки студенти або програмісти можуть навчитися:

Використовувати Flex для розробки лексичного аналізатора, який розпізнає числові значення та математичні оператори.

Використовувати Bison для побудови парсера, який визначає синтаксис виразу та створює абстрактне синтаксичне дерево.

Побудувати AST для виразів і використовувати його для обчислення значення виразу.

Опис функціональності

Проект реалізує такі основні функції:

Лексичний аналізатор (lexer.l):

Розпізнає числові значення, математичні оператори (+, -, \*, /) і круглі дужки ( і ).

Ігнорує пробіли та перенос рядка, що не впливають на структуру виразу.

Використовує Flex для створення токенів, які передаються в парсер.

Синтаксичний аналізатор (parser.y):

Визначає граматику для обробки базових арифметичних виразів із додаванням, відніманням, множенням і діленням, а також використанням круглих дужок.

Створює абстрактне синтаксичне дерево (AST), в якому кожен вузол відповідає операції або числу.

Реалізовано пріоритет операцій і асоціативність для коректного виконання арифметичних обчислень.

Абстрактне синтаксичне дерево (AST):

Для вираження використовуються два типи вузлів:

NumberNode: вузол для збереження числових значень.

BinaryOpNode: вузол для зберігання арифметичних операцій (+, -, \*, /) із посиланням на лівого та правого нащадків.

Кожен вузол має метод evaluate(), який обчислює результат виразу, ґрунтуючись на значеннях або операціях у вузлах.

Основна програма (main.cpp):

Викликає парсер для розбору введеного виразу.

Використовує кореневий вузол AST для обчислення кінцевого результату.

Виводить обчислене значення виразу або повідомлення про помилку, якщо вираз некоректний.

Makefile:

Автоматизує процес компіляції всіх файлів проекту.

Містить правила для компіляції lexer.l і parser.y, а також для лінкування всіх об'єктних файлів у виконуваний файл.

Основні кроки виконання програми

Програма очікує введення арифметичного виразу від користувача.

Лексичний аналізатор (Flex) розпізнає числові значення та оператори, повертаючи відповідні токени парсеру.

Парсер (Bison) будує абстрактне синтаксичне дерево (AST) на основі граматики для арифметичних виразів.

Після побудови AST основна програма обчислює значення виразу, використовуючи методи evaluate() у вузлах дерева.

Результат обчислення виводиться на екран.

Приклад роботи

Користувач вводить вираз, наприклад: 3 + 4 \* (2 - 1).

Програма розбирає та аналізує вираз, враховуючи пріоритет операцій та дужки.

Результат обчислення виводиться: Результат: 7.

Переваги та особливості проекту

Структурованість: Проект розділено на окремі модулі для лексичного та синтаксичного аналізу, а також для обробки AST, що робить його легким для розширення.

AST для обчислень: Використання абстрактного синтаксичного дерева дозволяє зручно обчислювати результат виразу та реалізувати більш складні математичні вирази в майбутньому.

Автоматизація збірки: Makefile забезпечує автоматичну компіляцію та полегшує керування залежностями між компонентами проекту.

Використані технології

Flex для лексичного аналізу, що перетворює текст на токени.

Bison для синтаксичного аналізу та побудови AST на основі заданої граматики.

C++ для реалізації основного коду та обробки результатів обчислення.

Це завдання на розробку синтаксичного аналізатора на основі Lex/YACC (або аналогів), призначеного для граматики мови програмування на базі C чи іншої імперативної мови, або для арифметичних виразів. Вам потрібно буде створити базову версію синтаксичного аналізатора, що може бути реалізована за допомогою Bison (GNU YACC) для синтаксичного аналізу та Flex для лексичного аналізу.

Нижче наведено базовий приклад, як реалізувати синтаксичний аналізатор для арифметичних виразів:

**Крок 1: Лексичний аналізатор (lexer.l)**

У файлі lexer.l змін не потрібно, оскільки він вже сумісний з C++.

%{

#include "parser.tab.h"

#include <cstdlib>

%}

%%

[0-9]+ { yylval = atoi(yytext); return NUMBER; }

[ \t\n] ; // ігнорувати пробіли та переноси рядків

"+" { return PLUS; }

"-" { return MINUS; }

"\*" { return MULT; }

"/" { return DIV; }

"(" { return LPAREN; }

")" { return RPAREN; }

. { printf("Невідомий символ: %s\n", yytext); }

%%

**Крок 2: Синтаксичний аналізатор (parser.y)**

Тут ми адаптуємо синтаксичний аналізатор до C++ та додаємо класи для побудови дерева виразів.

%{

#include <iostream>

#include <memory>

void yyerror(const char \*s);

int yylex();

struct ASTNode {

virtual ~ASTNode() = default;

virtual int evaluate() const = 0;

};

struct NumberNode : ASTNode {

int value;

explicit NumberNode(int v) : value(v) {}

int evaluate() const override { return value; }

};

struct BinaryOpNode : ASTNode {

std::unique\_ptr<ASTNode> left;

std::unique\_ptr<ASTNode> right;

char op;

BinaryOpNode(std::unique\_ptr<ASTNode> l, std::unique\_ptr<ASTNode> r, char o)

: left(std::move(l)), right(std::move(r)), op(o) {}

int evaluate() const override {

switch (op) {

case '+': return left->evaluate() + right->evaluate();

case '-': return left->evaluate() - right->evaluate();

case '\*': return left->evaluate() \* right->evaluate();

case '/': return left->evaluate() / right->evaluate();

default: throw std::runtime\_error("Невідома операція");

}

}

};

std::unique\_ptr<ASTNode> root;

%}

%token NUMBER

%token PLUS MINUS MULT DIV

%token LPAREN RPAREN

%left PLUS MINUS

%left MULT DIV

%%

expr:

expr PLUS expr { $$ = new BinaryOpNode($1, $3, '+'); }

| expr MINUS expr { $$ = new BinaryOpNode($1, $3, '-'); }

| expr MULT expr { $$ = new BinaryOpNode($1, $3, '\*'); }

| expr DIV expr { $$ = new BinaryOpNode($1, $3, '/'); }

| LPAREN expr RPAREN { $$ = $2; }

| NUMBER { $$ = new NumberNode($1); }

;

%%

void yyerror(const char \*s) {

std::cerr << "Помилка: " << s << std::endl;

}

int main() {

std::cout << "Введіть вираз: ";

if (yyparse() == 0 && root) {

std::cout << "Результат: " << root->evaluate() << std::endl;

}

return 0;

}

**Опис змін**

1. **Структури даних AST**:
   * Створено базову структуру ASTNode для представлення вузлів дерева. У неї є чисто віртуальна функція evaluate().
   * NumberNode представляє число (лист дерева).
   * BinaryOpNode представляє бінарні оператори (+, -, \*, /) і містить два дочірні вузли (left і right), а також оператор (op).
2. **Синтаксичні правила**:
   * Кожен вираз повертає вузол AST. Наприклад, для виразу expr PLUS expr створюється BinaryOpNode, який об'єднує лівий і правий вирази з оператором +.
3. **Глобальний корінь дерева**:
   * Змінна root містить корінь дерева AST, яке створюється під час парсингу. Після парсингу evaluate() викликається для обчислення результату виразу.

**Крок 3: Компіляція та виконання**

1. Згенеруємо файли для лексичного та синтаксичного аналізаторів:

flex lexer.l

bison -d parser.y

1. Скомпілюємо з C++ компілятором:

g++ lex.yy.c parser.tab.c -o parser -lfl -std=c++11

1. Запускаємо програму:

./parser

**Коментарі до адаптації:**

* **C++ особливості**: Використання std::unique\_ptr забезпечує автоматичне звільнення пам'яті для вузлів AST.
* **Оцінка дерева**: Використання evaluate() дозволяє обчислювати значення виразу, обходячи дерево з кореня.

**Крок 4: Створення main.cpp**

Файл main.cpp містить основну функцію для запуску парсера та обчислення результату.

// main.cpp

#include <iostream>

#include "parser.tab.h"

#include "ast.h"

extern std::unique\_ptr<ASTNode> root;

extern int yyparse();

int main() {

std::cout << "Введіть вираз: ";

if (yyparse() == 0 && root) {

try {

int result = root->evaluate();

std::cout << "Результат: " << result << std::endl;

} catch (const std::runtime\_error& e) {

std::cerr << "Помилка обчислення: " << e.what() << std::endl;

}

}

return 0;

}

**Крок 5: Компіляція проекту**

Створюємо Makefile, щоб полегшити процес компіляції всіх компонентів проекту:

# Makefile

all: parser

parser: lexer.o parser.o main.o

g++ -o parser lexer.o parser.o main.o -lfl -std=c++11

lexer.o: lexer.l parser.tab.h

flex lexer.l

g++ -c lex.yy.c -o lexer.o

parser.o: parser.y

bison -d parser.y

g++ -c parser.tab.c -o parser.o

main.o: main.cpp ast.h

g++ -c main.cpp -o main.o

clean:

rm -f parser lex.yy.c parser.tab.c parser.tab.h \*.o

**Крок 6: Запуск програми**

1. Виконуємо команду для компіляції:

make

1. Запускаємо програму:

./parser

1. Вводимо арифметичний вираз, наприклад:

scss

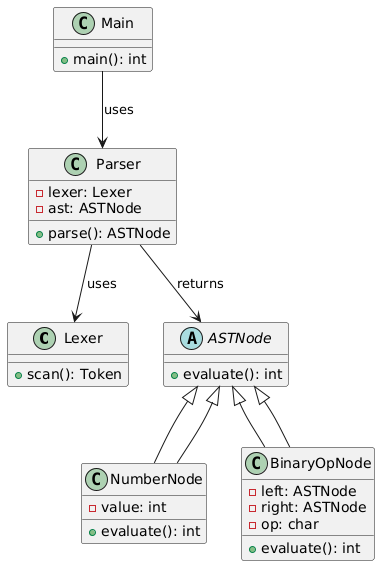
Копировать код

3 + 4 \* (2 - 1)

Програма виведе обчислений результат.

**Коментарі**

* **Обробка помилок**: В BinaryOpNode реалізовано обробку помилки для операції ділення на нуль.
* **Використання AST**: Структура AST використовується для зберігання й обчислення значення виразу.
* **Makefile**: Використання Makefile спрощує процес компіляції всіх файлів і дозволяє швидко очистити згенеровані файли.



UML Діаграма проекту

**Висновок**

Проект демонструє принципи роботи синтаксичних аналізаторів та обчислення значень виразів, побудову абстрактного синтаксичного дерева, використання лексичних та синтаксичних аналізаторів для побудови мовних інтерпретаторів і простих компіляторів.