#include <iostream>

#include <vector>

#include <cctype>

#include <string>

#include <memory>

#include <stdexcept>

enum TokenType {

TOKEN\_NUMBER,

TOKEN\_PLUS,

TOKEN\_MINUS,

TOKEN\_MULTIPLY,

TOKEN\_DIVIDE,

TOKEN\_LPAREN,

TOKEN\_RPAREN,

TOKEN\_END

};

struct Token {

TokenType type;

double value;

};

class Lexer {

public:

Lexer(const std::string& input) : input(input), pos(0) {}

Token getNextToken() {

while (pos < input.size() && isspace(input[pos])) pos++;

if (pos >= input.size()) return { TOKEN\_END, 0 };

char currentChar = input[pos];

if (isdigit(currentChar) || currentChar == '.') {

return number();

}

switch (currentChar) {

case '+': pos++; return { TOKEN\_PLUS, 0 };

case '-': pos++; return { TOKEN\_MINUS, 0 };

case '\*': pos++; return { TOKEN\_MULTIPLY, 0 };

case '/': pos++; return { TOKEN\_DIVIDE, 0 };

case '(': pos++; return { TOKEN\_LPAREN, 0 };

case ')': pos++; return { TOKEN\_RPAREN, 0 };

default: throw std::runtime\_error("Unknown token");

}

}

private:

Token number() {

std::string numStr;

while (pos < input.size() && (isdigit(input[pos]) || input[pos] == '.')) {

numStr += input[pos++];

}

return { TOKEN\_NUMBER, std::stod(numStr) };

}

std::string input;

size\_t pos;

};

struct ASTNode {

virtual ~ASTNode() = default;

};

struct NumberNode : ASTNode {

double value;

NumberNode(double value) : value(value) {}

};

struct BinaryOpNode : ASTNode {

std::unique\_ptr<ASTNode> left;

std::unique\_ptr<ASTNode> right;

TokenType op;

BinaryOpNode(std::unique\_ptr<ASTNode> left, std::unique\_ptr<ASTNode> right, TokenType op)

: left(std::move(left)), right(std::move(right)), op(op) {}

};

class Parser {

public:

Parser(Lexer& lexer) : lexer(lexer), currentToken(lexer.getNextToken()) {}

std::unique\_ptr<ASTNode> parse() {

return expr();

}

private:

std::unique\_ptr<ASTNode> expr() {

auto node = term();

while (currentToken.type == TOKEN\_PLUS || currentToken.type == TOKEN\_MINUS) {

TokenType op = currentToken.type;

currentToken = lexer.getNextToken();

node = std::make\_unique<BinaryOpNode>(std::move(node), term(), op);

}

return node;

}

std::unique\_ptr<ASTNode> term() {

auto node = factor();

while (currentToken.type == TOKEN\_MULTIPLY || currentToken.type == TOKEN\_DIVIDE) {

TokenType op = currentToken.type;

currentToken = lexer.getNextToken();

node = std::make\_unique<BinaryOpNode>(std::move(node), factor(), op);

}

return node;

}

std::unique\_ptr<ASTNode> factor() {

if (currentToken.type == TOKEN\_NUMBER) {

auto node = std::make\_unique<NumberNode>(currentToken.value);

currentToken = lexer.getNextToken();

return node;

}

else if (currentToken.type == TOKEN\_LPAREN) {

currentToken = lexer.getNextToken();

auto node = expr();

if (currentToken.type != TOKEN\_RPAREN) {

throw std::runtime\_error("Missing closing parenthesis");

}

currentToken = lexer.getNextToken();

return node;

}

else {

throw std::runtime\_error("Unexpected token");

}

}

Lexer& lexer;

Token currentToken;

};

class Evaluator {

public:

double evaluate(const std::unique\_ptr<ASTNode>& node) {

if (auto numberNode = dynamic\_cast<NumberNode\*>(node.get())) {

return numberNode->value;

}

else if (auto binOpNode = dynamic\_cast<BinaryOpNode\*>(node.get())) {

double left = evaluate(binOpNode->left);

double right = evaluate(binOpNode->right);

switch (binOpNode->op) {

case TOKEN\_PLUS: return left + right;

case TOKEN\_MINUS: return left - right;

case TOKEN\_MULTIPLY: return left \* right;

case TOKEN\_DIVIDE: return left / right;

default: throw std::runtime\_error("Unknown binary operator");

}

}

else {

throw std::runtime\_error("Unknown AST node");

}

}

};

int main() {

std::string input;

std::cout << "Enter an expression: ";

std::getline(std::cin, input);

try {

Lexer lexer(input);

Parser parser(lexer);

auto ast = parser.parse();

Evaluator evaluator;

double result = evaluator.evaluate(ast);

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

}

catch (const std::exception& e) {

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

}

return 0;

}