Team Meeting 11/20
WHEN: Wednesday, November 20th, 5:00 pm
PURPOSE: Discussion/Finalization of Parser, discussion of combining all components and interface implementation
ATTENDANCE: Kristoffer Comahig, Drew Franke, Gael Salazar-Morales, Quinn Westrope, Owen Berkholtz, Axel Bengoa, Bryson Toubassi

Agenda

- ☑ ~~Discussion/Finalization of Parser~~
- ☑ ~~Discussion of combining all components~~
- ☑ ~~Interface implementation~~
- ☑ ~~Test Cases / Error Handling~~

General Notes

In person meeting

Decided implementation for Parser (Kris)

Will handle simple interface implementation on day of Test Cases / Error Handling meeting

Everyone will create their own list of test cases through the Test Cases document and combine them into one document next week meeting (11/ 26)

Will deal with User Manual and Final Implementation Combination on 12/4

```
#include <iostream>
#include <string>
#include <cctype>
#include "lexer.h"
#include "ast.h"

treeNode* parseE(const std::string& input, size_t& pos);
treeNode* parseT(const std::string& input, size_t& pos);
treeNode* parseP(const std::string& input, size_t& pos);
treeNode* parseF(const std::string& input, size_t& pos);

Token prevToken;  //to keep track of past tokens for context (particularly for negation vs
subtraction)
Token nextToken;

treeNode* parseE(const string& input, size_t& pos) {
   treeNode* a = parseT(input, pos);
   while (true) {
         if (::nextToken.type == TokenType::TOKEN_PLUS) {
            if (prevToken.type == TokenType::TOKEN_NUMBER || prevToken.type ==
TokenType::TOKEN_RPAREN) {
               // It's addition (binary)
               prevToken = nextToken;
               nextToken = scanToken(input, pos);
               treeNode* b = parseT(input, pos);
               a = new Add(a, b);
            }
            else {
               // It's positive unary
               prevToken = nextToken;
               nextToken = scanToken(input, pos);
               treeNode* b = parseF(input, pos);
               a = new Plus(b);
            }
         }
         else if (::nextToken.type == TokenType::TOKEN_MINUS) { //might cause problems with
negation but we'll get to it later
```

```
            if (prevToken.type == TokenType::TOKEN_NUMBER || prevToken.type ==
TokenType::TOKEN_RPAREN) {
                // It's subtraction (binary)
                prevToken = nextToken;
                nextToken = scanToken(input, pos);
                treeNode* b = parseT(input, pos);
                a = new Sub(a, b);
            }
            else {
                // It's negation (unary)
                prevToken = nextToken;
                nextToken = scanToken(input, pos);
                treeNode* b = parseF(input, pos);
                a = new Negate(b);
            }
        }
        else {
            return a;
            break;
        }
    }
}

treeNode* parseT(const string& input, size_t& pos) {
    treeNode* a = parseP(input, pos);
    while (true) {
        if (::nextToken.type == TokenType::TOKEN_STAR) {
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            treeNode* b = parseP(input, pos);
            a = new Multi(a, b);
        }
        else if (::nextToken.type == TokenType::TOKEN_SLASH) {
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            treeNode* b = parseP(input, pos);
            a = new Div(a, b);
        }
        else if (::nextToken.type == TokenType::TOKEN_MODULO) {
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            treeNode* b = parseP(input, pos);
            a = new Mod(a, b);
        }
```

```
        else {
            return a;
            break;
        }
    }

}

treeNode* parseP(const string& input, size_t& pos) {
    treeNode* a = parseF(input, pos);
    while (true) {
        if (::nextToken.type == TokenType::TOKEN_EXPONENT) {  // Check for exponentiation
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            treeNode* b = parseF(input, pos);  // Exponentiation is applied to factors
            a = new Expo(a, b);  // Assuming Power is a node class for exponentiation
        }
        else {
            return a;
        }
    }
}

treeNode* parseF(const string& input, size_t& pos) {
    while (true) {
        if (::nextToken.type == TokenType::TOKEN_MINUS) {
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            if (::nextToken.type == TokenType::TOKEN_NUMBER) {
                treeNode* b = parseF(input, pos);
                return new Negate(b);
            }
            else if (::nextToken.type == TokenType::TOKEN_LPAREN) {
                prevToken = nextToken;
                nextToken = scanToken(input, pos);
                treeNode* b = parseE(input, pos);
                if (::nextToken.type == TokenType::TOKEN_RPAREN) {
                    prevToken = nextToken;
                    nextToken = scanToken(input, pos);
                    return new Negate(b);
                }
            }
        }
        else if (::nextToken.type == TokenType::TOKEN_PLUS) {
```

```
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            if (::nextToken.type == TokenType::TOKEN_NUMBER) {
                treeNode* b = parseF(input, pos);
                return new Plus(b);
            }
            else if (::nextToken.type == TokenType::TOKEN_LPAREN) {
                prevToken = nextToken;
                nextToken = scanToken(input, pos);
                treeNode* b = parseE(input, pos);
                if (::nextToken.type == TokenType::TOKEN_RPAREN) {
                    prevToken = nextToken;
                    nextToken = scanToken(input, pos);
                    return new Plus(b);
                }
            }
        }
        else if (::nextToken.type == TokenType::TOKEN_NUMBER) {
            float num = std::stof(::nextToken.value);
            treeNode* a = new Integer(num);
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            return a;
        }
        else if (::nextToken.type == TokenType::TOKEN_LPAREN) {
            prevToken = nextToken;
            nextToken = scanToken(input, pos);
            treeNode* a = parseE(input, pos);
            if (a == NULL) { return NULL; }
            if (::nextToken.type == TokenType::TOKEN_RPAREN) {
                prevToken = nextToken;
                nextToken = scanToken(input, pos);
                return a;
            }
            else { return NULL; break; }
        }
        else {
            return NULL;
        }
    }
}
```

```
int main() {
    size_t pos = 0;
    string input = "+(-2) * (-3) - ((-4) / (+5))";
    //-(+2) * (+3) - (-4) / (-5)
    nextToken = scanToken(input, pos);
    prevToken.type = TokenType::TOKEN_START; //sets previous token as start, to avoid
recursive issues
    treeNode* resultTree = parseE(input, pos);

    std::cout << resultTree->getValue() << std::endl;
}
```

```
treeNode* parseE();
treeNode* parseT();
treeNode* parseF();

treeNode* parseE() {
    // Parse the first term of the expression
    treeNode* left = parseT();

    // Continue parsing while the next token is either a plus or minus
operator.
    while (nextToken.type == TOKEN_PLUS || nextToken.type == TOKEN_MINUS)
{
        // Store the operator type.
        TokenType op = nextToken.type;

        // Scan the next token.
        nextToken = scanToken(input, pos, nextToken);

        // Parse the next term of the expression.
        treeNode* right = parseT();

        // If the operator is a plus, create an addition node.
        if (op == TOKEN_PLUS) {
            left = new Add(left, right);
        // If the operator is a minus, create a subtraction node.
        } else if (op == TOKEN_MINUS) {
            left = new Sub(left, right);
```

```
        }
    }

    // Return the root of the parsed expression tree.
    return left;
}

// Parses a term in the expression grammar
treeNode* parseT() {
    // Parse the first factor
    treeNode* left = parseF();
    while (nextToken.type == TOKEN_STAR || nextToken.type == TOKEN_SLASH)
{
        TokenType op = nextToken.type;
        nextToken = scanToken(input, pos, nextToken);
        treeNode* right = parseF();
        if (op == TOKEN_STAR) {
            left = new Multi(left, right);
        } else if (op == TOKEN_SLASH) {
            left = new Div(left, right);
        }
    }
    return left;
}

// Parses a factor in the expression grammar
treeNode* parseF() {
    if (nextToken.type == TOKEN_NUMBER) {
        // Convert the token value to an integer
        int value = std::stoi(nextToken.value);
        // Move to the next token
        nextToken = scanToken(input, pos, nextToken);
        // Return a new Integer node with the parsed value
        return new Integer(value);
    } else if (nextToken.type == TOKEN_MINUS) {
        // Move to the next token
        nextToken = scanToken(input, pos, nextToken);
        // Return a new Negate node with the parsed factor
        return new Negate(parseF());
    } else if (nextToken.type == TOKEN_LPAREN) {
```

```
        // Move to the next token
        nextToken = scanToken(input, pos, nextToken);
        // Parse the enclosed expression
        treeNode* expr = parseE();
        if (nextToken.type == TOKEN_RPAREN) {
            // Move to the next token
            nextToken = scanToken(input, pos, nextToken);
        }
        // Return the parsed expression
        return expr;
    }
    // Return nullptr if no valid factor is found
    return nullptr;
```

```
#include "lexer_Drewf.h"
#include "ast.h"
#include <memory>
#include <stdexcept>


class Parser {
public:
    // Get first token from the lexer to start parsing
    Parser(Lexer& lexer) : lexer(lexer) {

    currentToken = getNextToken();
}

    treeNode* parseExpression() {
        return addORsubtract();
    }

private:

    Lexer& lexer;
    Token currentToken;

    // Get the next token
    Token getNextToken() {
        std::optional<Token> nextToken = lexer.scanToken();
        if (nextToken) {
```

```
            return *nextToken;
        } else {
            return Token{"END", ""};
        }
    }

    // function to move on to the next token
    void move_to_next_token(const std::string& input) {
        if (currentToken.type == input) {
            currentToken = getNextToken();
        } else {
            throw std::runtime_error("Unexpected token: " + currentToken.value + ", expected: " +
input);
        }
    }


    // add or subtract
    treeNode* addORsubtract() {

        // check higher level of precedence
        treeNode* node = multORdivORmod();

        while (currentToken.type == "PLUS" || currentToken.type == "MINUS") {
            std::string op = currentToken.type;
            if (op == "PLUS") {
                move_to_next_token("PLUS");
                node = new Add(node, multORdivORmod());
            }
            else if (op == "MINUS") {
                move_to_next_token("MINUS");
                node = new Sub(node, multORdivORmod());
            }
        }

        return node;
    }

    // multiply, divide, or modulo
    treeNode* multORdivORmod() {

        // check higher level of precedence
        treeNode* node = exponentiate();
```

```
        while (currentToken.type == "MULTIPLY" || currentToken.type == "DIVIDE" ||
currentToken.type == "MOD") {
            std::string op = currentToken.type;
            if (op == "MULTIPLY") {
                move_to_next_token("MULTIPLY");
                node = new Multi(node, exponentiate());
            }
            else if (op == "DIVIDE") {
                move_to_next_token("DIVIDE");
                node = new Div(node, exponentiate());
            }
            else if (op == "MOD") {
                move_to_next_token("MOD");
                node = new Modulo(node, exponentiate());
            }
        }

        return node;
    }

    // exponents
    treeNode* exponentiate() {

        // check higher level of precedence
        treeNode* node = unary();

        while (currentToken.type == "EXPONENT") {
            move_to_next_token("EXPONENT");
            node = new Exponent(node, exponentiate()); // Right associativity
        }

        return node;
    }

    // Unary operators
    treeNode* unary() {
        if (currentToken.type == "PLUS") {
            move_to_next_token("PLUS");
            return unary(); // Unary plus can be ignored
        }
        else if (currentToken.type == "MINUS") {
            move_to_next_token("MINUS");
            return new Negate(unary());
        }
```

```
        else {
            return primary();
        }
    }

    // highest level of precedence
    treeNode* primary() {
        if (currentToken.type == "NUMBER") {
            int value = std::stoi(currentToken.value);
            move_to_next_token("NUMBER");
            return new Integer(value);
        }
        else if (currentToken.type == "LPAREN") {
            move_to_next_token("LPAREN");
            treeNode* node = addORsubtract();
            move_to_next_token("RPAREN");
            return node;
        }
        else {
            throw std::runtime_error("Invalid token: " + currentToken.value);
        }
    }
};
```