

Woke Mathematical Expression Parser

Anthony Tendwa Michael

Woke Mathematical Expression Parser

Introduction

I used lexing and parsing to develop a customized calculator capable of handle complicated mathematical equations with user-defined operators, all while injecting it with modern woke and Gen Z lingo instead of traditional symbols for my mini-project. This project demonstrates the actual application of these principles, demonstrating their revolutionary potential in the development of real-world solutions. While context-free grammars provide a theoretical framework for parsing, they are only used indirectly through the parsing logic. This project not only demonstrates a thorough mastery of lexing and parsing, but also how these key ideas smoothly mix theory with hands-on problem-solving, bridging the gap between classroom learning and practical application.

Parsing the Path to Precision

In this section, I'll go over the fine intricacies and reasoning that went into the creation of the custom calculator. This project's major purpose is to enable the calculator to process mathematical formulas utilizing custom operators represented by woke and Gen Z words. I've constructed a unique and functional custom calculator that uses woke and Gen Z words for arithmetic operations by implementing these components and using the parsing and lexing ideas, bringing a different approach on calculator architecture.

Let's break down the key elements and underlying reasoning that I employed in this endeavor:

I. Lexing

The lexing step was where I started the project. Raw user input is converted into a stream of tokens during lexing. I created a set of regular expressions to recognize and classify numerical data such as integers and floating-point numbers. To handle special operators like 'add-up', 'split-the-bill', 'smash,' 'slay,' 'flex,' and 'glow-up,' I created separate regular expressions to identify these phrases. This methodical methodology ensures that even the most unusual operators are appropriately tokenized.

II. Parsing

The parsing step is at the heart of the project, decoding tokens and generating an abstract syntax tree (AST) for evaluation. To execute the parsing logic, I created a recursive descent parser, a top-down parsing approach. The parsing mechanism employs a set of rules based on context-free grammars to navigate the custom operators' associativity and precedence levels. This guarantees that the mathematical expressions are evaluated correctly and in the correct sequence.

III. Operator Actions

I assigned certain actions to each operator. When an operator token is encountered, the parser performs the relevant operation, such as addition, subtraction, multiplication, division, percentage computation, or exponentiation. These actions are rigorously crafted to produce correct outcomes while taking into account the unique semantics of each operator.

IV. Error Handling

To meet multiple eventualities, the project features strong error handling. When the calculator encounters unexpected or unknown tokens, it displays clear and useful error messages. Error-handling logic that has been meticulously crafted improves the overall user experience by guiding users through the right input format.

Relevance of Course Concepts in Real Applications

This project graphically depicts the relationship between the generated solution and the fundamental principles learned during the course. One notable relationship is the parsing approach used. To parse mathematical statements, the project implements a left-to-right recursive descent parsing technique, a notion that is commonly taught in class.

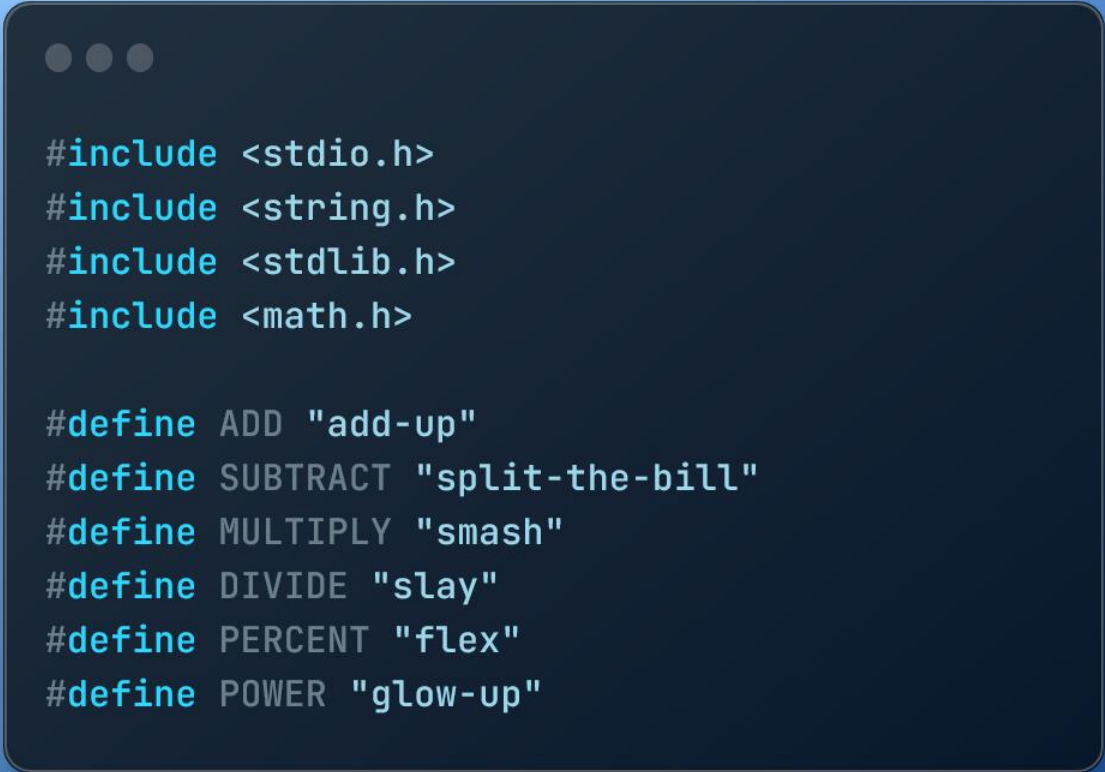
This parsing mechanism allows the calculator to process expressions sequentially, from left to right. It should be noted that this specific parsing technique does not automatically follow the conventional sequence of operations. The project, on the other hand, exhibits the versatility of parser logic because it is meant to enforce the order defined by custom operators, such as 'smash' taking priority over 'add-up.'

This adaptability shows the adaptability and usefulness of parsing ideas, demonstrating how they may be tailored to individual requirements. Furthermore, it demonstrates the course topics' usefulness in real-world applications, stressing the importance and significance of parsing and lexing in software development.

Dive into the Code

I. Include Libraries and Define Constants

In this code block, we include the necessary standard libraries for input/output, string manipulation, memory allocation, and math operations. We also define constants for custom operators like 'add-up,' 'split-the-bill,' 'smash,' 'slay,' 'flex,' and 'glow-up.' These constants will be used later for identifying and performing specific operations.



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <math.h>

#define ADD "add-up"
#define SUBTRACT "split-the-bill"
#define MULTIPLY "smash"
#define DIVIDE "slay"
#define PERCENT "flex"
#define POWER "glow-up"
```

II. Operator Calculation Function

This function, `performOperation`, takes two operands and an operator as arguments and performs the corresponding mathematical operation based on the operator. It checks the operator type using `strcmp` and returns the result of the operation.

```
double performOperation(double num1, char* operator, double num2) {
    if (strcmp(operator, ADD) == 0) return num1 + num2;
    if (strcmp(operator, SUBTRACT) == 0) return num1 - num2;
    if (strcmp(operator, MULTIPLY) == 0) return num1 * num2;
    if (strcmp(operator, DIVIDE) == 0) {
        if (num2 == 0) {
            printf("Error: Division by zero\n");
            exit(1);
        }
        return num1 / num2;
    }
    if (strcmp(operator, PERCENT) == 0) return num1 * (num2 / 100.0);
    if (strcmp(operator, POWER) == 0) return pow(num1, num2);

    printf("Error: Unknown operator '%s'\n", operator);
    exit(1);
}
```

III. Main Program Loop

This part of the code implements the main program loop, allowing the user to enter and calculate mathematical expressions. It repeatedly asks for user input, tokenizes the input, and processes the tokens to perform calculations. The loop continues until the user chooses to exit.


```

int main() {
    while (1) {
        char input[1000];
        printf("Enter an expression: ");
        fgets(input, sizeof(input), stdin);

        char* token = strtok(input, " \n");
        double result = 0.0;
        int isResultSet = 0;

        while (token != NULL) {
            if (strcmp(token, ADD) == 0 || strcmp(token, SUBTRACT) == 0 ||
                strcmp(token, MULTIPLY) == 0 || strcmp(token, DIVIDE) == 0 ||
                strcmp(token, PERCENT) == 0 || strcmp(token, POWER) == 0) {
                char* operator = token;
                token = strtok(NULL, " \n");
                if (token != NULL) {
                    double operand = atof(token);
                    result = performOperation(result, operator, operand);
                    isResultSet = 1;
                } else {
                    printf("Error: Operator '%s' requires an operand\n", operator);
                    break;
                }
            } else if (atof(token) != 0 || strcmp(token, "0") == 0) {
                if (isResultSet) {
                    printf("Error: Expected an operator, found '%s'\n", token);
                    break;
                }
                result = atof(token);
                isResultSet = 1;
            } else {
                printf("Error: Unrecognized token '%s'\n", token);
                break;
            }
            token = strtok(NULL, " \n");
        }

        if (isResultSet) printf("Result: %.1f\n");

        char response;
        printf("Do you want to calculate again? (y/n): ");
        scanf(" %c", &response);
        if (response != 'y' && response != 'Y') break;
        getchar();
    }

    return 0;
}

```

Conclusion

Finally, this example shows how lexing and parsing may be used in the context of a customized calculator. Using bespoke operators symbolized by current slang terminology like 'add-up', 'split-the-bill', 'smash', 'slay', 'flex', and 'glow-up', this project provides a fresh viewpoint on mathematical expression evaluation. Despite not strictly following the standard sequence of operations, this calculator effectively parses expressions, handles custom operators, and executes calculations while remaining connected to the essential principles learnt in class. It demonstrates the flexibility of these key principles in contemporary programming and language design by illustrating how classic notions in programming and parsing may be changed and applied in a modern, user-friendly, and interesting manner.