CIÊNCIAS DA COMPUTAÇÃO

FACULDADE DE CIÊNCIAS | UNIVERSIDADE AGOSTINHO NETO

PROGRAMAÇÃO IMPERATIVA

TEMA

ANALISADOR E AVALIADOR DE EXPRESSÕES MATEMÁTICAS

DOCENTE

Amândio de Jesus Almada

Luanda Junho de 2025

CIÊNCIAS DA COMPUTAÇÃO

FACULDADE DE CIÊNCIAS | UNIVERSIDADE AGOSTINHO NETO

TEMA

ANALISADOR E AVALIADOR DE EXPRESSÕES MATEMÁTICAS

INTEGRANTES DO GRUPO:

- **Vandelson Malebo:** vandelsonmalebo02@gmail.com **/ 945 056 121**
- **Razão Bernardo Nzundo:** ----------@uan.co.ao/ **944 578 953**
- **Jone Imperial:** imperialjone@gmail.com / **949 355 120 – 957 523 245**
- **Rosanio Sardinha:** zidanepoke20@gmail.com/ **926 230 519**

ÍNDICE

1. INTRODUÇÃO

No universo da programação, lidar com expressões numéricas é uma tarefa fundamental, mas cuja complexidade muitas vezes passa despercebida. Ao digitar uma expressão como (10 - 5) * 3 em um programa, espera-se que o resultado correto seja retornado automaticamente, sem refletir sobre os mecanismos internos que possibilitam esse processamento. Entretanto, por trás dessa simplicidade aparente, existe um processo essencial conhecido como análise de expressões, responsável por interpretar e converter essas sentenças matemáticas em instruções compreensíveis pelo computador.

A análise de expressões é um dos pilares na construção de compiladores, interpretadores e diversos outros sistemas que lidam com cálculos. O desafio principal reside na correta interpretação da precedência de operadores, no tratamento de parênteses e na conversão de strings para valores numéricos.

1.1. Apresentação breve do projeto

Este projeto tem como objetivo a implementação de um avaliador de expressões aritméticas em linguagem C, capaz de ler expressões matemáticas a partir de um arquivo de entrada (in.txt), validá-las sintaticamente e calcular seus respectivos resultados, escrevendo-os em um arquivo de saída (out.txt).

O funcionamento do programa baseia-se em três etapas principais:

1. **Tokenização** – a expressão é dividida em componentes fundamentais (números, operadores e parênteses);
2. **Validação** – a estrutura da expressão é analisada, verificando parênteses balanceados, presença de operadores e operandos em posições corretas, além de prevenir divisões por zero;
3. **Conversão e Avaliação** – expressões válidas são convertidas da notação infixa para pós-fixa (notação polonesa reversa) e, em seguida, avaliadas utilizando uma pilha para cálculo do resultado final.

Durante a execução, o programa também detecta e trata diversos tipos de erros, como uso de caracteres inválidos, sintaxe incorreta, parênteses desbalanceados e divisão por zero.

A construção deste analisador permite ao aluno compreender de forma prática como funciona o processo de análise e avaliação de expressões numéricas em linguagens de programação, reproduzindo parte da lógica utilizada por compiladores e interpretadores.

1.2. Explicação do contexto

Este projeto foi desenvolvido com o intuito de proporcionar uma experiência prática e educativa, permitindo nos estudantes explorar os fundamentos da análise de expressões aritméticas. Através da construção de um avaliador próprio, é entender como uma expressão matemática escrita em notação infixa como 3 + (4 * 2) pode ser transformada e avaliada pelo computador de forma estruturada.

Além disso, o projeto foi pensado para reforçar e explorar conceitos fundamentais da linguagem C, como manipulação de strings, estruturas de dados (pilhas, fila), leitura e escrita em arquivos, e organização modular do código. Ao final, o estudante é capaz de compreender na prática os processos de **tokenização**, **validação sintática**, **conversão para notação pós-fixa** e **avaliação de expressões**, reproduzindo etapas fundamentais do funcionamento de um compilador real.

1.3. Objetivo principal

O principal objetivo deste projeto é desenvolver um programa em linguagem C capaz de ler, validar, interpretar e calcular expressões aritméticas fornecidas por meio de um arquivo de entrada, apresentando os resultados ou mensagens de erro em um arquivo de saída.

2. DESCRIÇÃO DO PROJETO

O projeto consiste na criação de um programa em linguagem C que realiza a leitura de expressões aritméticas armazenadas em um arquivo de entrada (in.txt), analisa sua validade estrutural e calcula o resultado de forma automatizada, registrando as saídas valores numéricos ou mensagens de erro em um arquivo de saída (out.txt).

A lógica do programa está dividida em etapas bem definidas:

1. **Leitura de Arquivo:** O programa inicia verificando se os arquivos in.txt e out.txt foram corretamente passados como argumentos. Em seguida, realiza a leitura linha por linha do arquivo de entrada.

2. **Tokenização:** Cada linha lida é dividida em tokens, que representam os componentes fundamentais da expressão, como números, operadores (+, -, *, /, ^) e parênteses. Durante essa etapa, caracteres inválidos são detectados e geram erro.

3. **Validação da Expressão:** Com base nos tokens, o programa verifica se a estrutura da expressão é válida.

4. **Conversão para Notação Pós-fixa:** Expressões válidas em notação infixa são convertidas para notação pós-fixa utilizando uma pilha, respeitando as regras de precedência e associatividade dos operadores.

5. **Avaliação da Expressão:** A expressão em notação pós-fixa é avaliada utilizando uma pilha de operandos. O resultado final é calculado passo a passo, aplicando os operadores binários sobre os valores.

6. **Saída dos Resultados:** Após o processamento, os resultados corretos ou mensagens de erro específicas (como "Divisão por zero" ou "Expressão malformada") são gravados no arquivo out.txt, além de exibidos no console para acompanhamento visual.

O projeto utiliza estruturas como Token e Pilha, implementadas em arquivos separados (ex: dstlib_include.h), promovendo organização modular do código e facilitando a manutenção e leitura.

2.1. Objetivos Específicos

Para alcançar o objetivo principal do projeto, foram definidos os seguintes objetivos específicos:

- **Implementar um sistema de leitura de expressões aritméticas** a partir de um arquivo de entrada (in.txt), processando linha por linha de forma sequencial.
- **Realizar a tokenização das expressões**, separando números, operadores e parênteses em unidades léxicas (tokens) válidas para posterior análise.
- **Validar a sintaxe das expressões**, garantindo que:
    - Os parênteses estejam corretamente balanceados;
    - A ordem entre operadores e operandos esteja correta;
    - Não ocorram divisões por zero;
    - Caracteres inválidos sejam detectados e tratados.
- **Converter expressões da notação infixa para pós-fixa** (notação reversa), respeitando a precedência e associatividade dos operadores aritméticos.
- **Avaliar corretamente as expressões em notação pós-fixa**, utilizando pilhas para armazenar operandos e operadores temporários durante o cálculo.
- **Gerar um arquivo de saída (out.txt)**, contendo os resultados das expressões válidas e mensagens de erro específicas para as expressões inválidas.
- **Aplicar os conceitos fundamentais da linguagem C**, como manipulação de arquivos,

3. METODOLOGIA E DESENVOLVIMENTO

A metodologia aplicada inclui um levantamento bibliográfico fundamentado em obras de referência e publicações científicas, bem como a análise de artigos, investigações em sites e blogs acadêmicos.

A implementação do projeto foi feita em linguagem C, utilizando o paradigma procedural e a modularização por meio de arquivos de cabeçalho (headers), visando facilitar a organização e reutilização do código.

3.1 Algoritmos e Estruturas de Dados Utilizados

Algoritmos

- **Tokenização**: Utilizado um algoritmo de varredura de caracteres (análise léxica).
- **Validação Sintática.**
- **Conversão Infixa para Pós-fixa (*Shunting Yard Algorithm*)**.
- **Avaliação Pós-fixa**: Utilizado um algoritmo com pilha.

Estruturas de Dados

- **Token**: Representa cada elemento da expressão (número, operador ou parêntese), com campos para tipo, valor e peso.
- **Pilha**: Estrutura essencial utilizada para:
    - Armazenar operadores durante a conversão infixa → pós-fixa.
    - Armazenar operandos durante a avaliação pós-fixa.

3.2 Estratégia de Modularização

O projeto foi dividido em múltiplos arquivos para manter clareza e organização:

- **main.c**: Contém a função principal e o controle de fluxo geral do programa (leitura, processamento e escrita).
- **dstlib_include.h**: Contém a definição da estrutura Token, Pilha, tipos e constantes auxiliares.

3.3 Dificuldades Encontradas e Como Foram Superadas

Durante a implementação do projeto, algumas dificuldades surgiram:

1. Validação sintática: Implementar uma validação que detectasse todos os casos de expressões malformadas, como parênteses desbalanceados, operadores em posições inválidas ou divisões por zero.

Solução:

- Utilizamos um contador de parênteses para verificar balanceamento
- Implementamos um sistema de "expectativa" (esperar_operando) para validar a sequência correta de tokens
- Verifica explicitamente divisões por zero durante a validação

2. Tokenização: Processar corretamente a string de entrada, identificando e separando os diferentes tipos de tokens (números, operadores, parênteses) enquanto ignorava espaços em branco.

Solução: Desenvolvemos a função tokenizar() que:

- Percorre a string caractere por caractere
- Usamos funções como isdigit() para identificar números multi-dígitos
- Classificamos cada token corretamente e valida operadores desconhecidos

3. Conversão para Notação Pós-fixa: Implementar o algoritmo shunting-yard para conversão infixa-pósfixa, considerando precedência de operadores e associatividade (especialmente a exponenciação).

Solução: Na função infixa_para_posfixa():

- Utilizemos uma pilha para armazenar operadores temporariamente
- Implementamos regras específicas para cada tipo de token
- Tratamos a associatividade à direita do operador '^' de forma diferente dos outros operadores

4. Avaliação de Expressões Pós-fixas: Avaliar corretamente a expressão pós-fixa, lidando com erros como falta de operandos ou divisões por zero que só são detectáveis durante a avaliação.

Solução: Na função avaliar_posfixa():

- Utilizemos uma pilha para armazenar operandos
- Implementamos verificações de erro para operandos insuficientes
- Tratamos a divisão por zero como um caso especial durante a avaliação

4. RESULTADOS E TESTES

Durante a execução do projeto, foram realizados diversos testes com expressões válidas e inválidas no arquivo in.txt, a fim de verificar os resultados do programa, sua precisão nos cálculos e a capacidade de identificar e tratar erros. Os resultados observados permitiram avaliar o funcionamento geral do programa.

Todas as funcionalidades foram implementadas com sucesso e estão a funcionar bem sem problema.

4.1 Recomendação Futura

- **Suporte a Números Decimais**

  Atualmente só funciona com inteiros

  Poderia ser estendido para números de ponto flutuante

- **Operadores Unários**

  Não há suporte para operadores unários como +5 ou -3

  Requereria modificações no tokenizador e no validador

CONCLUSÃO

O desenvolvimento deste projeto proporcionou uma experiência valiosa para compreender, de forma prática, como expressões aritméticas podem ser analisadas, validadas e avaliadas por um programa de computador.

Através da construção de um analisador em linguagem C, foi possível implementar com sucesso todas as etapas necessárias: desde a leitura e tokenização das expressões, passando pela validação sintática rigorosa, até a conversão para notação pós-fixa e avaliação correta dos resultados.

O programa demonstrou ser capaz de identificar diferentes tipos de erros (como parênteses desbalanceados, divisão por zero, e uso de caracteres inválidos), gerando respostas adequadas tanto em tela quanto no arquivo de saída. As expressões válidas foram processadas corretamente, com o resultado final calculado e registrado conforme esperado.

Além dos resultados técnicos alcançados, o projeto proporcionou importantes aprendizados, tais como:

- A aplicação de estruturas de dados, como **pilhas**, em contextos reais de processamento matemático;
- A prática de boa **modularização** e organização de código em C;
- A importância do **tratamento de erros** e da validação de entrada;
- A experiência em trabalhar com **manipulação de arquivos** e **estruturação de funções**.

Em suma, o projeto cumpriu seu objetivo gerais e especifíco entendendo o funcionamento interno de interpretadores.

REFERÊNCIAS BIBLIOGRÁFICAS

Algoritmo Shunting Yard em C: implementação completa e comentada em C, com explicação de parsing e RPN: “Shunting yard algorithm (C)” (LiterateProograms) stackoverflow.com+4literateprograms.org+4en.wikipedia.org+4

Algoritmo Shunting Yard (via Rosetta Code) : mostra passos do algoritmo de forma didática e exemplificada: “Parsing/Shunting-yard algorithm” (Rosetta Code) geeksforgeeks.org+9rosettacode.org+9github.com+9

Utilização de strtok() para tokenização : excelente referência para quebra de strings em C, útil para etapas de leitura: “String Tokenization in C” (GeeksforGeeks) reddit.com+12geeksforgeeks.org+12stackoverf

Linguagem C, Luís Dramas, tradução de João Araújo Ribeiro, Orlando Bernardo Filho, - 10ed. – Rio de Janeiro: LTC, 2007.

Link do Github: https://github.com/vandelsonmalebo02/Projecto_PI-G06.git