Sumário

1. Introdução:

A resolução de expressões matemáticas é uma tarefa fundamental em computação e em diversos campos da ciência e engenharia. Uma das formas mais eficientes de representar e resolver expressões matemáticas é através da notação pós-fixa (ou pós-fixada), também conhecida como notação polonesa reversa.

Neste documento, vamos abordar o processo de transformação de uma expressão matemática infixa para a notação pós-fixa e, em seguida, demonstrar como avaliar essa expressão pós-fixa.

Uma vez que a expressão tenha sido convertida para a notação pós-fixa, podemos avaliá-la utilizando uma estrutura de dados conhecida como pilha. A notação pós-fixa oferece uma maneira eficiente de representar e resolver expressões matemáticas, eliminando a ambiguidade da ordem das operações, e assim tornando ela mais parecida com a forma em que se é resolvida manualmente, mas de forma automatizada em C. Pode ser encontrado no link a seguir.

GitHub: https://github.com/FilipiNyetz/ProjetoCalculadora/blob/main/calculadora.c

2. Implementação:

Arquivo: calculadora.h

Neste arquivo calculadora.h contém as declarações de uma estrutura de dados chamada Expressao, que representa uma expressão matemática.

- **posFixa**: Uma string que armazena a expressão na forma pós-fixa, por exemplo, "3 12 4 + *".
- **inFixa**: Uma string que armazena a expressão na forma infixada, por exemplo, "3 * (12 + 4)".
- **Valor**: Um valor numérico que representa o resultado da avaliação da expressão.

Além disso, o arquivo contém protótipos de duas funções:

- **getFormaInFixa**: Uma função que recebe uma string como entrada e retorna a forma infixada da expressão (posFixa).
- **getValor**: Uma função que calcula o valor numérico da expressão (na forma pós-fixa) e retorna o resultado como um número em ponto flutuante.

Arquivo: calculadora.c

Neste arquivo é iniciado o papel de criar as funções que fortalecem o conceito de conversão de dados, como int para string, e também o uso de operadores lógicos, e alguns mais complexos como Sen, Cos, e Log.

- Este arquivo contém a implementação das funções para manipular expressões matemáticas e realizar operações como conversão entre notações e avaliação de expressões.
- A função **getValor** recebe uma expressão matemática pós-fixa como entrada e retorna o resultado da avaliação da expressão.
- Durante a avaliação, a função utiliza uma pilha para armazenar e processar os operandos e operadores da expressão, realizando as operações conforme a ordem pós-fixa.
- A função **ehOperador** verifica se um caractere é um operador matemático.
- A função **precedencia** retorna a precedência de um operador.
- A função **ehFuncaoMatematica** verifica se uma string representa uma função matemática como log, seno ou cosseno.
- A função **getFormaInFixa** recebe uma expressão matemática pós-fixa como entrada e retorna a mesma expressão convertida para a forma infixada.
- Durante a conversão, a função utiliza uma pilha para reorganizar os operadores e operandos, e manipula parênteses conforme a precedência dos operadores.

Essas funções fornecem uma interface para manipular expressões matemáticas e realizar operações como conversão entre notações e avaliação de expressões.

3. Testes

Nesta seção, juntamente à imagem descrita abaixo, poderá ser observado o resultado dos testes das expressões escolhidas e descritas nos subtítulos de numeração “3.1” a “3.7” para a realização de testes necessários para a verificação da funcionalidade das funções que realizam o cálculo das expressões inseridas, bem como as funções que realizam a conversão das expressões pós-fixas para infixas.

Pode-se notar na figura abaixo (Figura 01) que as expressões elaboradas foram criadas visando abranger todas as operações matemáticas que esse programa é capaz de realizar (soma, subtração, multiplicação, divisão, exponenciação, logaritmo, seno e cosseno). No teste conduzido abaixo, também foi possível testar a total funcionalidade do programa desenvolvido (calculadora de expressões pós-fixas), ou seja, pôde-se verificar se todas as funções criadas e usadas no programa estavam funcionando da forma desejada, como as funções responsáveis por gerenciar a pilha de dados, escrita e conversão de expressões, verificador de operações, a de resultado da expressão matemática e todas as outras funções presentes e descritas neste documento e no próprio programa. Observe a saída do programa:

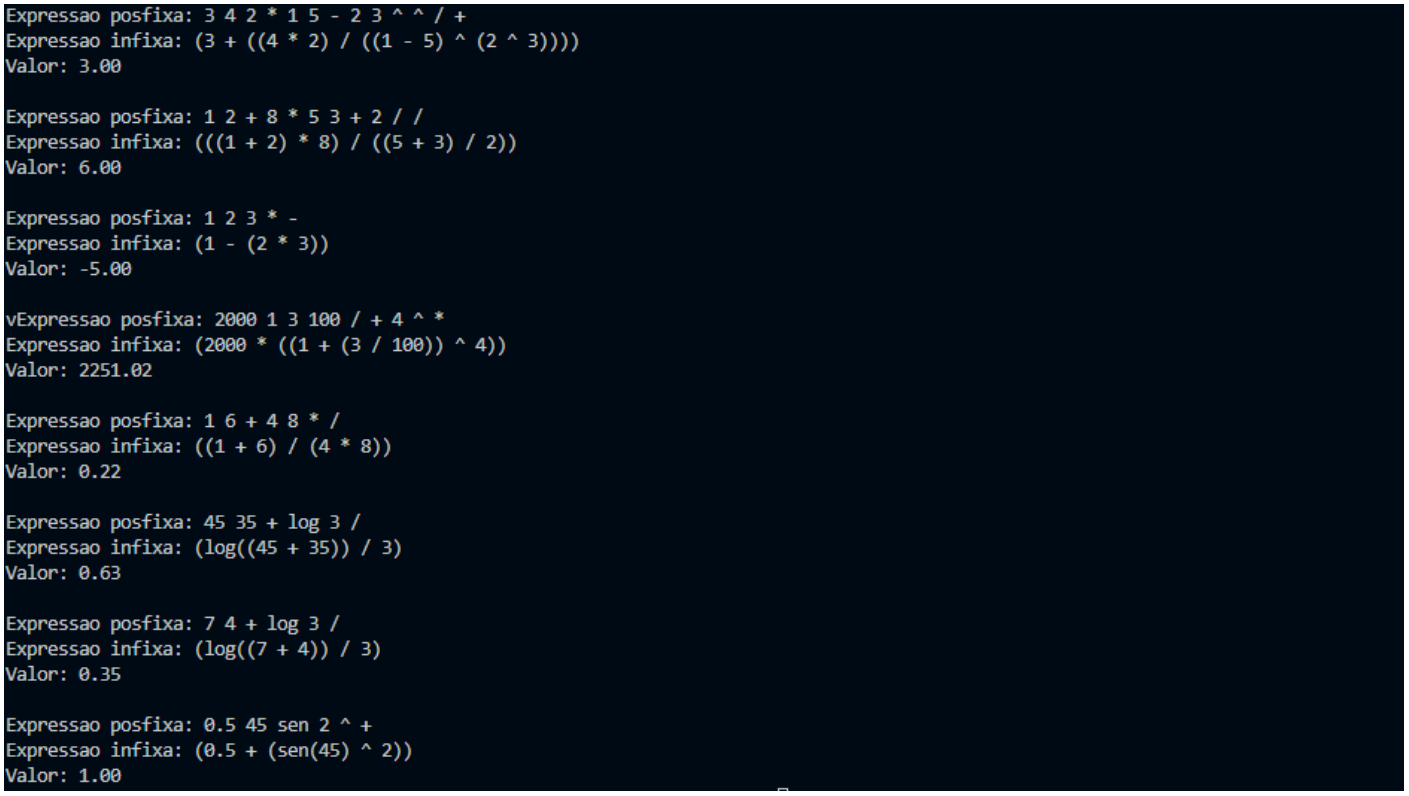

Figura 01 - Teste de Funcionalidades Gerais

No teste que poderá ser observado na Figura 03, foi feita a verificação da capacidade do programa para a leitura de expressões fora da formatação que o programa foi desenvolvido para ler. Pôde-se concluir que a aplicação possui boa tolerância ao ainda ser capaz de realizar o cálculo corretamente, contudo, o programa é incapaz de organizar as expressões (tanto infixas quanto pós-fixas) inseridas com formatação indevida. Apesar da compilação e execução perfeita do código, a saída do programa apresenta esses erros indicados aqui neste parágrafo. Veja a formatação indevidas das expressões usadas, na Figura 02, e observe a saída na Figura 03:

```
char posFixa1[] = "3   4  2*1 5-2 3 ^^ /+";
char posFixa2[] = "1    2   +8*   5 3+2/   /";
char posFixa3[] = "1   2 3   *-";
```

Figura 02 - Expressões inseridas com formatação indevida

```
Expressao posfixa: 3   4  2*   1 5-  2 3 ^^ /+
Expressao infixa: ((3 * 4) + ((2 - 1) / (5 ^ (2 ^ 3))))
Valor: 3.00

Expressao posfixa: 1    2   +8*   5 3+2/   /
Expressao infixa: ((3 / (3)
Valor: 6.00

Expressao posfixa: 1   2 3   *-
Expressao infixa: (1 - (2 * 3))
Valor: -5.00
```

Figura 03 - Saída ao inserir expressões com formatação indevida

No teste a seguir, com a saída representada pela Figura 04, é apresentado um erro causado pela não identificação da definição das funções responsáveis para calcular logarítmo, seno e cosseno de valores inseridos nas expressões. Esse teste foi conduzido no momento de desenvolvimento do código da funcionalidade desses cálculos matemáticos citados e que, como já observado, já teve seu erro reparado antes do lançamento desta aplicação para avaliação do professor orientador. Veja a saída do programa ao tentar fazer sua compilação naquele momento:

```
/home/estevaolins/Área de Trabalho/calculadora/calculadora.c:42: referência não definida para "log10"
/usr/bin/ld: /home/estevaolins/Área de Trabalho/calculadora/calculadora.c:47: referência não definida para "sin"
/usr/bin/ld: /home/estevaolins/Área de Trabalho/calculadora/calculadora.c:52: referência não definida para "cos"
```

Figura 04 - Saída apontando os erros de referência não definidas para os cálculos matemáticos de logarítmo, seno e cosseno.

3.1 Expressão 3 4 2 * 1 5 - 2 3 ^ ^ / +

O valor da expressão 3 4 2 * 1 5 - 2 3 ^ ^ / +, na forma infixa, é 3 + 4 * 2 / (1 - 5) ^ 2 ^ 3, tem valor igual a 3.00012207 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 3 e empilha.	**[3]**
2	Lê 4 e empilha.	[3, 4]
3	Lê 2 e empilha.	[3, 4, 2]
4	Lê *, desempilha os últimos valores, calcula 4 * 2 = 8, e empilha 8.	[3, 8]
5	Lê 1 e empilha.	[3, 8, 1]
6	Lê 5 e empilha.	[3, 8, 1, 5]
7	Lê -, desempilha os últimos valores, calcula 1 - 5 = -4, e empilha -4.	[3, 8, -4]
8	Lê 2 e empilha.	[3, 8, -4, 2]
9	Lê 3 e empilha.	[3, 8, -4, 2, 3]
10	Lê ^, desempilha os últimos valores, calcula 2 ^ 3 = 8, e empilha 8.	[3, 8, -4, 8]
11	Lê ^, desempilha os últimos valores, calcula (-4) ^ 8 = 65536, empilhando-o.	[3, 8, 65536]
12	Lê /, desempilha 8 e 65536, calcula 8 / 65536 = 0.00012207, empilhando-o.	[3, 0.00012207]
13	Lê +, desempilha os últimos operandos e efetua cálculos, empilhando o resultado.	[3.00012207]

3.2 Expressão 1 2 + 8 * 5 3 + 2 //

O valor da expressão 1 2 + 8 * 5 3 + 2 // na forma infixa (((1 + 2) * 8) / ((5 + 3) / 2)), tem valor igual a 6 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 1 e empilha.	**[1]**
2	Lê 2 e empilha.	[1, 2]
3	Lê +, desempilha os valores, calcula 1+2=3, e empilha 3 .	[3]
4	Lê 8 e empilha.	[3, 8]
5	Lê *, desempilha os valores, calcula 3*8=24, e empilha 24 .	[24]
6	Lê 5 e empilha.	[24,5]
7	Lê 3 e empilha.	[24,5,3]
8	Lê +, desempilha os últimos valores, calcula 5+3=8, e empilha 8 .	[24,8]
9	Lê 2 e empilha.	[24,8,2]
10	Lê /, desempilha os últimos valores, calcula 8/2 = 4, e empilha 4.	[24,4]
11	Lê /, desempilha os últimos valores, calcula 24/4 = 6, e empilha o resultado.	[6]

3.3 Expressão 1 2 3 *

O valor da expressão 1 2 3 * -, na forma infixa 1 - (2*3) tem valor igual a -5 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 1 e empilha.	**[1]**
2	Lê 2 e empilha.	[1, 2]
3	Lê 3 e empilha.	[1,2,3]
4	Lê *, desempilha os valores, calcula 2*3=6, e empilha 6 .	[1,6]
5	Lê -, desempilha os últimos valores, calcula 1-6 = -5, e empilha o resultado.	[-5]

3.4 Expressão 2000 1 3 100 / + 4 ^ *

O valor da expressão 2000 1 3 100 / + 4 ^ *, na forma infixa (2000 * ((1 + (3 / 100)) ^ 4)) tem valor igual a 2251 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 2000 e empilha.	**[2000]**
2	Lê 1 e empilha.	[2000,1]
3	Lê 3 e empilha.	[2000,1,3]
4	Lê 100 e empilha.	[2000,1,3,100]
5	Lê /, desempilha os valores, calcula 3/100=0.03, e empilha 0.03 .	[2000,1,0.03]
6	Lê +, desempilha os valores, calcula 1+0.03, e empilha 1.03 .	[2000,1.03]
7	Lê 4 e empilha.	[2000,1.03,4]
8	Lê ^, desempilha os últimos valores, calcula 1.03^ 4=1.255, e empilha 1.1255 .	[2000,1.1255]

9	Lê *, desempilha os últimos valores, calcula 2000*1.255 = 25100, e empilha o resultado.	[2251]

3.5 Expressão 1 6 + 4 8 * /

O valor da expressão 1 6 + 4 8 * /, na forma infixa ((1 + 6) / (4 * 8)) tem valor igual a 0.22 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 1 e empilha.	**[1]**
2	Lê 6 e empilha.	[1, 6]
3	Lê +, desempilha os valores, calcula 1+6=7, e empilha 7 .	[7]
4	Lê 4 e empilha.	[7, 4]
5	Lê 8 e empilha	[7, 4, 8]
6	Lê * , calcula 4 * 8 = 32, e empilha o 32	[7, 32]
7	Lê /, desempilha os últimos valores, calcula 7/32=0.22, e empilha o resultado.	[0.22]

3.6 Expressão 45 35 + log 3 /

O valor da expressão 45 35 + log 3 /, na forma infixa (log((45 + 35)) / 3) tem valor igual a 0.63 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 45 e empilha.	**[45]**
2	Lê 35 e empilha.	[45, 35]
3	Lê +, desempilha os valores, calcula 45+35=80, e empilha 80 .	[80]
4	Lê log e, desempilha os valores, calcula log(80),e empilha 1.90.	[1.90308998699]
5	Lê /, e calcula 1.90/3, e empilha o resultado,	[0.63]

3.7 Expressão 7 4 + log 3 /

O valor da expressão 7 4 + log 3 /, na forma infixa (log((7 + 4)) / 3) tem valor igual a 0.35 e pode ser obtido a partir do detalhamento apresentado na tabela.

		Pilha
1	Lê 7 e empilha.	**[7]**
2	Lê 4 e empilha.	[7, 4]
3	Lê +, desempilha os valores, calcula 7+4=11, e empilha 11 .	[11]
4	Lê log, desempilha o valor, calcula log(11)=1.05, e empilha 1.05.	[1.05]
5	Lê 3 e empilha.	[1.05, 3]
6	Lê /, desempilha os últimos valores, calcula 1.05/3=0.35, e empilha o resultado.	[0.35]

4. Conclusão

As expressões polonesas, notação concisa e clara para representar operações matemáticas, oferecem diversas vantagens em diferentes áreas. Implementá-las em C, linguagem de alto desempenho e controle preciso da memória, traz grandes resultados. Eficiência e Desempenho: C garante processamento rápido e otimizado de expressões matemáticas, utilizando manipulação direta de memória e recursos de strings. Integração com Sistemas Existentes: C facilita a integração com sistemas embarcados e aplicações de tempo real, permitindo comunicação com outras linguagens e sistemas. Flexibilidade e Adaptabilidade: C permite personalização da lógica de avaliação e otimização para diferentes plataformas, com recursos para tratamento de erros e exceções. Aprendizagem e Crescimento Profissional: A implementação em C aprimora habilidades como manipulação de memória, algoritmos e resolução de problemas, além de aprofundar o conhecimento em matemática e computação. Combinando a simplicidade da notação polonesa com o poder e flexibilidade do C, essa implementação torna-se uma ferramenta poderosa para o desenvolvimento de aplicações matemáticas, lógicas e computacionais robustas e escaláveis

Referências

https://petbcc.ufscar.br/math/ *-> fala sobre lib de math*

https://www.geeksforgeeks.org/c-library-string-h/ *-> fala sobre lib de string*

Anexos

calculadora.h

```
#ifndef EXPRESSAO_H
#define EXPRESSAO_H

typedef struct {
    char posFixa[512];     // Expressão na forma pos fixa, como 3 12 4 + *
    char inFixa[512];      // Expressão na forma pos fixa, como 3 * (12 + 4)
    float Valor;           // Valor numérico da expressão
} Expressao;

char *getFormaInFixa(char *Str);    // Retorna a forma inFixa de Str (posFixa)
float getValor(char *Str);          // Calcula o valor de Str (na forma posFixa)

#endif
```

calculadora.c

```
#include <stdio.h>

#include <stdlib.h>

#include <ctype.h>

#include <string.h>

#include <math.h>

#include "calculadora.h"


#define MAX 512




float getValor(char *Str) {

    double pilha[MAX];

    int topo = -1;

    char buffer[100]; // Buffer para armazenar números temporariamente

    int j = 0; // Índice para o buffer
```

```
    for (int i = 0; Str[i] != '\0'; ++i) {
        if (isdigit(Str[i]) || Str[i] == '.') {
            // Se for um dígito ou ponto decimal, armazena no buffer
            buffer[j++] = Str[i];
        } else if (Str[i] == ' ' || Str[i] == '\t') {
            // Se for um espaço, converte o buffer para double e empilha
            if (j > 0) {
                buffer[j] = '\0'; // Termina a string
                pilha[++topo] = strtod(buffer, NULL);
                j = 0; // Reseta o índice do buffer
            }
        } else {
            // Se for um operador, reseta o buffer e empilha o número
            if (j > 0) {
                buffer[j] = '\0';
                pilha[++topo] = strtod(buffer, NULL);
                j = 0;
            }

                // Se for um operador, desempilha os operandos necessários e
realiza a operação
            double operando1, operando2;
            double valor = 0;

            if (strncmp(&Str[i], "log", 3) == 0) {
                operando1 = pilha[topo--];
```

```
            valor = log10(operando1);
            pilha[++topo] = valor;
            i += 2; // Pula sobre 'log'
        } else if (strncmp(&Str[i], "sen", 3) == 0) {
            operando1 = pilha[topo--] * 0.01744;
            valor = sin(operando1);
            pilha[++topo] = valor;
            i += 2; // Pula sobre 'sen'
        } else if (strncmp(&Str[i], "cos", 3) == 0) {
            operando1 = pilha[topo--] * 0.01744;
            valor = cos(operando1);
            pilha[++topo] = valor;
            i += 2; // Pula sobre 'cos'
        } else {
            operando2 = pilha[topo--];
            operando1 = pilha[topo--];

            switch (Str[i]) {
                case '+':
                    pilha[++topo] = operando1 + operando2;
                    break;
                case '-':
                    pilha[++topo] = operando1 - operando2;
                    break;
                case '*':
                    pilha[++topo] = operando1 * operando2;
```

```
                        break;
                    case '/':
                        if (operando2 != 0.0) {
                            pilha[++topo] = operando1 / operando2;
                        } else {
                            printf("Erro: Divisão por zero\n");
                            exit(EXIT_FAILURE);
                        }
                        break;
                    case '^':
                        pilha[++topo] = pow(operando1, operando2);
                        break;
                    default:
                        printf("Erro: Operador inválido\n");
                        exit(EXIT_FAILURE);
                }
            }
        }
    }
    // O resultado final estará no topo da pilha
    return pilha[topo];
}

int ehOperador(char c) {
    return (c == '+' || c == '-' || c == '*' || c == '/' || c == '^');
}
```

```
// Função para retornar a precedência de um operador

int precedencia(char operador) {

    if (operador == '^')

        return 3;

    else if (operador == '*' || operador == '/')

        return 2;

    else if (operador == '+' || operador == '-')

        return 1;

    else

        return 0; // Operador inválido

}




int ehFuncaoMatematica(char *str) {

      return (strncmp(str, "log", 3) == 0 || strncmp(str, "sen", 3) == 0 ||
strncmp(str, "cos", 3) == 0);

}




char *getFormaInFixa(char *Str) {

    static char resposta[MAX];

    char pilha[MAX][MAX];

    int topo = -1;

      char buffer[MAX]; // Buffer para armazenar temporariamente números ou
operadores

    int j = 0; // Índice para o buffer
```

```
    for (int i = 0; Str[i] != '\0'; ++i) {
        if (isdigit(Str[i]) || Str[i] == '.') {
            // Se for um dígito ou ponto decimal, armazena no buffer
            buffer[j++] = Str[i];
        } else if (Str[i] == ' ' || Str[i] == '\t') {
            // Se for um espaço, converte o buffer para string e empilha
            if (j > 0) {
                buffer[j] = '\0'; // Termina a string
                strcpy(pilha[++topo], buffer);
                j = 0; // Reseta o índice do buffer
            }
        } else if (ehOperador(Str[i])) {
                // Se for um operador, desempilha os operandos necessários e
realiza a operação
            char operando2[MAX], operando1[MAX], resultado[MAX];

            // Desempilha os operandos
            strcpy(operando2, pilha[topo--]);
            strcpy(operando1, pilha[topo--]);

            // Verifica a precedência e necessidade de parênteses
            int prec_op = precedencia(Str[i]);
            int prec_op1 = precedencia(operando1[0]);
            int prec_op2 = precedencia(operando2[0]);
                   int precisa_parenteses = (prec_op > prec_op1 || prec_op >
prec_op2);
```

```
        // Constrói a expressão infixa
        resultado[0] = '\0';
        if (precisa_parenteses) {
            strcat(resultado, "(");
            strcat(resultado, operando1);
            strcat(resultado, " ");
            strncat(resultado, &Str[i], 1);
            strcat(resultado, " ");
            strcat(resultado, operando2);
            strcat(resultado, ")");
        } else {
            strcat(resultado, operando1);
            strcat(resultado, " ");
            strncat(resultado, &Str[i], 1);
            strcat(resultado, " ");
            strcat(resultado, operando2);
        }

        // Empilha o resultado
        strcpy(pilha[++topo], resultado);

    } else if (ehFuncaoMatematica(&Str[i])) {
        // Funções como log, sen, cos
        char operando[MAX], resultado[MAX];
        char func[4]; // Buffer para armazenar a função (log, sen, cos)
```

```
            // Desempilha o operando
            strcpy(operando, pilha[topo--]);

            // Constrói a expressão infixa
            strncpy(func, &Str[i], 3);
            func[3] = '\0'; // Termina a string da função
            sprintf(resultado, "%s(%s)", func, operando);

            // Avança o índice da string para pular a função
            i += 2; // Pula sobre os dois caracteres adicionais da função

            // Empilha o resultado
            strcpy(pilha[++topo], resultado);
        }
    }

    // A expressão infixa final estará no topo da pilha
    strcpy(resposta, pilha[topo]);
    return resposta;
}
```

main.c

```c
int main() {


    char posFixa[] = "53 23 + 8 2 - *";

    // Convertendo para a forma infixa

    char *inFixa = getFormaInFixa(posFixa);

    // Calculando o valor da expressão

    float valor = getValor(posFixa);

    return 0;
}
```