Sumário

Introdução:

O presente exercício prático tem como objetivo desenvolver um sistema computacional, utilizando a linguagem de programação C, para analisar, converter e avaliar expressões matemáticas. O programa implementado tem como principal funcionalidade interpretar expressões numéricas tanto em notação infixada quanto em notação pós-fixada, permitindo ainda a conversão entre essas duas formas e o cálculo do valor resultante da expressão.

Problema a Ser Tratado:

Na matemática computacional e em linguagens de programação, a manipulação de expressões matemáticas requer mecanismos eficientes para avaliação e conversão de formatos. A notação infixada, embora mais natural para humanos, não é ideal para computadores avaliarem diretamente. Já a notação pós-fixada é mais adequada para implementação em máquinas por evitar o uso de parênteses e seguir uma ordem clara de execução. O desafio consiste em construir um sistema que entenda ambas as notações, converta entre elas corretamente e ainda realize o cálculo final, considerando operadores binários, funções matemáticas e regras de precedência.

Objetivos:

Avaliar expressões matemáticas, escritas na forma pós-fixada, realizando corretamente as operações aritméticas e funções matemáticas especiais.

Converter expressões, da forma infixada para pós-fixada e vice-versa, mantendo a coerência sintática e semântica.

Implementar suporte a operações binárias básicas e funções unárias.

Utilizar a estrutura de dados de pilha, fundamental para o processamento de expressões em notação pós-fixada, tanto para cálculo quanto para conversão.

GitHub:

https://github.com/10111213-up/Avalia-o-de-express-es-num-ricas

Implementação:

A principal estrutura de dados utilizada na implementação do sistema é a pilha, que se mostrou essencial para a conversão e avaliação de expressões matemáticas, especialmente na notação pós-fixada. Foram utilizadas duas variações de pilhas:

Pilha de números: Usada para armazenar operandos numéricos durante a avaliação das expressões pós-fixadas.

Estrutura:

typedef struct {

```
    float itens[MAX];

    int topo;

} PilhaFloat;
```

Pilha de strings: Utilizada durante a conversão de expressões entre infixa e pós-fixa, armazenando partes da expressão (números, operadores ou funções).

Estrutura:

```
typedef struct {

char* itens[MAX];

int topo;

} PilhaStr;
```

Protótipos de Funções

O programa possui diversas funções para suportar a conversão e avaliação de expressões. A seguir, estão os protótipos principais, definidos no arquivo expressao.h, com suas respectivas finalidades:

char *getFormaInFixa(char *Str)

Converte uma expressão da notação pós-fixada para infixada, usando pilhas de strings para reconstruir a forma tradicional da expressão com parênteses.

char *getFormaPosFixa(char *Str)

Converte uma expressão da forma infixada para pós-fixada, respeitando a precedência dos operadores e a estrutura dos parênteses.

float getValorPosFixa(char *StrPosFixa)

Avalia uma expressão matemática fornecida em notação pós-fixada, executando operações aritméticas e funções matemáticas, com uso de uma pilha de números.

float getValorInFixa(char *StrInFixa);

Avalia uma expressão infixada, convertendo-a internamente para a forma pós-fixada e, em seguida, utilizando a função getValorPosFixa para obter o valor numérico final.

1.1 Uso do TAD Pilha

O Tipo Abstrato de Dados Pilha foi um componente essencial na construção do avaliador de expressões numéricas. Ele foi usado para gerenciar operandos, operadores e expressões durante a conversão e avaliação, atuando em diferentes partes do programa.

Pilhas utilizadas:

O projeto implementou dois tipos distintos de pilhas:

- *Pilha de números (PilhaFloat): armazena valores numéricos (float) para a avaliação de expressões pós-fixadas.*
- *Pilha de strings (PilhaStr): armazena expressões parciais (char*) para a conversão da forma pós-fixada para infixada.*

Aplicações do TAD Pilha no projeto

Avaliação de expressões pós-fixadas

Função responsável: getValorPosFixa

Essa função percorre os tokens da expressão pós-fixada. Quando encontra um número, o empilha. Quando encontra um operador ou função, desempilha os operandos necessários, aplica a operação e empilha o resultado.

Conversão de pós-fixada para infixada

Função responsável: getFormaInFixa

Essa função utiliza a PilhaStr para remontar a expressão infixada a partir da pós-fixada. Quando encontra um número ou operando, empilha. Quando encontra um operador, desempilha dois elementos, os insere numa nova string com parênteses e empilha de volta.

Conversão de infixada para pós-fixada

Função responsável: getFormaPosFixa

Essa função converte uma expressão infixada para pós-fixada usando uma pilha auxiliar de operadores. Ela utiliza as regras de precedência para empilhar operadores e controlar parênteses.

As vantagens do uso do TAD Pilha são: organização e modularidade, cada pilha atende a uma função clara e separada. Eficiência na resolução da precedência de operadores e agrupamento de expressões. Facilidade de expansão, o uso de pilhas permite, no futuro, suportar mais operadores e tipos de expressões.

Testes

1.2 Teste n° 01

Entrada pós-fixa:

3 4 + 5 *

Processo na pilha:

Empilha 3

Empilha 4

Encontra + → Desempilha 3 e 4, calcula 3 + 4 = 7

Empilha 7

Empilha 5

Encontra * → Desempilha 7 e 5, calcula 7 * 5 = 35

Empilha 35

Saída infixa:

(3 + 4) * 5

Valor calculado: 35

1.3 Teste n° 02

Entrada pós-fixa:

*7 2 * 4 +*

Processo na pilha:

Empilha 7

Empilha 2

*Encontra * → Desempilha 7 e 2, calcula 7 * 2 = 14*

Empilha 14

Empilha 4

Encontra + → Desempilha 14 e 4, calcula 14 + 4 = 18

Saída infixa:

*7 * 2 + 4*

Valor calculado: 18

1.4 Teste n° 03

Entrada pós-fixa:

*8 5 2 4 + * +*

Processo na pilha:

Empilha 8

Empilha 5

Empilha 2

Empilha 4

Encontra + → Desempilha 2 e 4, calcula 2 + 4 = 6

Empilha 6

*Encontra * → Desempilha 5 e 6, calcula 5 * 6 = 30*

Empilha 30

Encontra + → Desempilha 8 e 30, calcula 8 + 30 = 38

Saída infixa:

*8 + (5 * (2 + 4))*

Valor calculado: 38

1.5 Teste n° 04

Entrada pós-fixa:

6 2 / 3 + 4 *

Processo na pilha:

Empilha 6

Empilha 2

Encontra / → Desempilha 6 e 2, calcula 6 / 2 = 3

Empilha 3

Empilha 3

Encontra + → Desempilha 3 e 3, calcula 3 + 3 = 6

Empilha 6

Empilha 4

Encontra * → Desempilha 6 e 4, calcula 6 * 4 = 24

Saída infixa:

(6 / 2 + 3) * 4

Valor calculado: 24

1.6 Teste n° 05

Entrada pós-fixa:

*9 5 2 8 * 4 + * +*

Processo na pilha:

Empilha 9

Empilha 5

Empilha 2

Empilha 8

*Encontra * → Desempilha 2 e 8, calcula 2 * 8 = 16*

Empilha 16

Empilha 4

Encontra + → Desempilha 16 e 4, calcula 16 + 4 = 20

Empilha 20

*Encontra * → Desempilha 20 e 5, calcula 20 * 5 = 100*

Empilha 100

Encontra + → Desempilha 100 e 9, calcula 100 + 9 = 109

Saída infixa:

*9 + (5 * (2 * 8 + 4))*

Valor calculado: 109

1.7 Teste n° 06

(Acrescente aqui uma expressão em notação pós-fixada e o processo de utilização da pilha, de forma similar ao que foi estabelecido no enunciado deste trabalho prático)

Pilha inicia vazia.

Lê 5 → número → empilha → Pilha: [5]

Lê 8 → número → empilha → Pilha: [5, 8]

Lê 4 → número → empilha → Pilha: [5, 8, 4]

Lê * → operador → desempilha 4 e 8, calcula 8 * 4 = 32, empilha → Pilha: [5, 32]

Lê + → operador → desempilha 32 e 5, calcula 5 + 32 = 37, empilha → Pilha: [37]

Lê 3 → número → empilha → Pilha: [37, 3]

Lê - → operador → desempilha 3 e 37, calcula 37 - 3 = 34, empilha → Pilha: [34]

Conclusão

Este trabalho teve como objetivo principal a construção de um sistema capaz de avaliar expressões numéricas, com suporte à conversão entre as formas infixada e pós-fixada, utilizando a linguagem de programação C e a estrutura de dados pilha como base. A proposta foi alcançada com sucesso: o sistema demonstrou ser capaz de processar corretamente expressões contendo operações aritméticas básicas e funções matemáticas especiais, como seno, cosseno, tangente, logaritmo e raiz quadrada.

Os resultados obtidos foram satisfatórios, especialmente nos testes realizados que confirmaram a precisão dos cálculos e a fidelidade das conversões entre as formas de notação. A correta utilização da pilha foi essencial para garantir que as expressões fossem tratadas de forma estruturada, respeitando a ordem das operações e a lógica de execução exigida pela notação pós-fixada.

Durante a implementação, algumas dificuldades foram enfrentadas, como:

- *O tratamento adequado das funções unárias durante a conversão e avaliação.*
- *A conversão de expressões com múltiplos níveis de parênteses, exigindo atenção à lógica de precedência.*
- *O controle e a manipulação de strings dinâmicas em C, o que exigiu cuidado com alocação e liberação de memória.*

Apesar dessas dificuldades, todas as funcionalidades previstas foram implementadas corretamente, e o sistema se mostrou estável e eficiente.

Como possíveis melhorias para versões futuras, destacam-se:

- *Suporte a mais funções matemáticas, como exponencial, logaritmo natural e funções hiperbólicas.*
- *Tratamento de erros sintáticos, como parênteses desbalanceados ou operadores inválidos.*
- *Implementação de uma interface gráfica simples ou versão em linha de comando interativa.*
- *Inclusão de suporte a variáveis e expressões armazenadas, para permitir reutilização de resultados.*

Em resumo, o trabalho foi bastante enriquecedor, tanto no aspecto técnico quanto conceitual, permitindo uma aplicação prática dos conhecimentos sobre estruturas de dados, manipulação de expressões e desenvolvimento em linguagem C.

Referências

MARCELO. Filas. Aula virtual da disciplina de Matemática Financeira. Universidade Católica. Ambiente Virtual de Aprendizagem

https://ava.catolica.edu.br/d2l/le/enhancedSequenceViewer/104995?url=https%3A%2F%2F211c9f77-18c9-42e7-a7d8-b67813cc574d.sequences.api.brightspace.com%2F104995%2Factivity%2F1921931%3FfilterOnDatesAndDepth%3D1.

MARCELO. Pilhas. Aula virtual da disciplina de Matemática Financeira. Universidade Católica. Ambiente Virtual de Aprendizagem

https://ava.catolica.edu.br/d2l/le/enhancedSequenceViewer/104995?url=https%3A%2F%2F211c9f77-18c9-42e7-a7d8-b67813cc574d.sequences.api.brightspace.com%2F104995%2Factivity%2F1926477%3FfilterOnDatesAndDepth%3D1

LIMA, Daniel. Pilhas e Filas: Estruturas de Dados e Implementação em Código. Apresentação em PowerPoint. SlideShare, 10 set. 2023. Disponível em: https://www.slideshare.net/daniel-lima/pilhas-e-filas-estrutura-de-dados. Acesso em: 20 jun. 2025.

Anexos

calculadora.h

```
#ifndef EXPRESSAO_H

#define EXPRESSAO_H



typedef struct {

   char posFixa[512];

   char inFixa[512];

   float Valor;

} Expressao;



char *getFormaInFixa(char *Str);

char *getFormaPosFixa(char *Str);

float getValorPosFixa(char *StrPosFixa);

float getValorInFixa(char *StrInFixa);
```

```
#endif
```

calculadora.c

```
#include <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <math.h>

#include "expressao.h"

#include <ctype.h>


#define MAX 512


typedef struct {

   float itens[MAX];

   int topo;

} PilhaFloat;


void initPilhaFloat(PilhaFloat* p) {

   p->topo = -1;

}


int vaziaFloat(PilhaFloat* p) {

   return p->topo == -1;

}
```

```
void pushFloat(PilhaFloat* p, float val) {
    p->itens[++p->topo] = val;
}

float popFloat(PilhaFloat* p) {
    return p->itens[p->topo--];
}

typedef struct {
    char* itens[MAX];
    int topo;
} PilhaStr;

void initPilhaStr(PilhaStr* p) {
    p->topo = -1;
}

void pushStr(PilhaStr* p, const char* str) {
    p->itens[++p->topo] = strdup(str);
}

char* popStr(PilhaStr* p) {
    return p->itens[p->topo--];
}
```

```
float grausParaRadianos(float graus) {
    return graus * M_PI / 180.0;
}

int isFuncao(const char* s) {
    return (
        strcmp(s, "sen") == 0 || strcmp(s, "cos") == 0 ||
        strcmp(s, "tg") == 0 || strcmp(s, "log") == 0 ||
        strcmp(s, "raiz") == 0
    );
}

int isOperador(const char* s) {
    return (
        strcmp(s, "+") == 0 || strcmp(s, "-") == 0 || strcmp(s, "*") == 0 ||
        strcmp(s, "/") == 0 || strcmp(s, "^") == 0 || strcmp(s, "%") == 0
    );
}

float getValorPosFixa(char* StrPosFixa) {
    PilhaFloat pilha;
    initPilhaFloat(&pilha);

    char* token = strtok(StrPosFixa, " ");
    while (token) {
```

```
        if (isOperador(token)) {
            float b = popFloat(&pilha);
            float a = popFloat(&pilha);
            if (strcmp(token, "+") == 0) pushFloat(&pilha, a + b);
            else if (strcmp(token, "-") == 0) pushFloat(&pilha, a - b);
            else if (strcmp(token, "*") == 0) pushFloat(&pilha, a * b);
            else if (strcmp(token, "/") == 0) pushFloat(&pilha, a / b);
            else if (strcmp(token, "^") == 0) pushFloat(&pilha, powf(a, b));
            else if (strcmp(token, "%") == 0) pushFloat(&pilha, fmodf(a, b));
        } else if (isFuncao(token)) {
            float a = popFloat(&pilha);
            if (strcmp(token, "sen") == 0) pushFloat(&pilha, sinf(grausParaRadianos(a)));
            else if (strcmp(token, "cos") == 0) pushFloat(&pilha, cosf(grausParaRadianos(a)));
            else if (strcmp(token, "tg") == 0) pushFloat(&pilha, tanf(grausParaRadianos(a)));
            else if (strcmp(token, "log") == 0) pushFloat(&pilha, log10f(a));
            else if (strcmp(token, "raiz") == 0) pushFloat(&pilha, sqrtf(a));
        } else {
            pushFloat(&pilha, atof(token));
        }
        token = strtok(NULL, " ");
    }
    return vaziaFloat(&pilha) ? 0.0 : popFloat(&pilha);
}

char* getFormaInFixa(char* Str) {
```

```
PilhaStr pilha;
initPilhaStr(&pilha);


char* token = strtok(Str, " ");
while (token) {
   if (isOperador(token)) {
      char* b = popStr(&pilha);
      char* a = popStr(&pilha);
      char buffer[MAX];
      snprintf(buffer, MAX, "(%s %s %s)", a, token, b);
      pushStr(&pilha, buffer);
      free(a); free(b);
   } else if (isFuncao(token)) {
      char* a = popStr(&pilha);
      char buffer[MAX];
      snprintf(buffer, MAX, "%s(%s)", token, a);
      pushStr(&pilha, buffer);
      free(a);
   } else {
      pushStr(&pilha, token);
   }
   token = strtok(NULL, " ");
}


return popStr(&pilha);
```

```
}

int ehOperador(char c) {
    return strchr("+-*/%^", c) != NULL;
}

int prioridade(char op) {
    switch (op) {
        case '+': case '-': return 1;
        case '*': case '/': case '%': return 2;
        case '^': return 3;
        default: return 0;
    }
}

char* getFormaPosFixa(char* Str) {
    static char saida[MAX];
    saida[0] = '\0';

    char* stack[MAX];
    int topo = -1;

    char token[MAX];
    int i = 0, j = 0;
```

```
while (Str[i] != '\0') {
    char c = Str[i];

    if (isdigit(c) || c == '.') {
        token[j++] = c;
    } else {
        if (j > 0) {
            token[j] = '\0';
            strcat(saida, token);
            strcat(saida, " ");
            j = 0;
        }

        if (isalpha(c)) {
            j = 0;
            while (isalpha(Str[i])) {
                token[j++] = Str[i++];
            }
            token[j] = '\0';

            stack[++topo] = strdup(token);
            j = 0;
            continue;
        }
```

```
    if (c == '(') {
        stack[++topo] = strdup("(");
    } else if (c == ')') {
        while (topo >= 0 && strcmp(stack[topo], "(") != 0) {
            strcat(saida, stack[topo--]);
            strcat(saida, " ");
        }
        if (topo >= 0) free(stack[topo--]);

        if (topo >= 0 && isFuncao(stack[topo])) {
            strcat(saida, stack[topo--]);
            strcat(saida, " ");
        }
    } else if (ehOperador(c)) {
        while (topo >= 0 && ehOperador(stack[topo][0]) &&
               prioridade(stack[topo][0]) >= prioridade(c)) {
            strcat(saida, stack[topo--]);
            strcat(saida, " ");
        }
        char* op = malloc(2);
        op[0] = c;
        op[1] = '\0';
        stack[++topo] = op;
    }
}
```

```
        i++;
    }

    if (j > 0) {
        token[j] = '\0';
        strcat(saida, token);
        strcat(saida, " ");
    }

    while (topo >= 0) {
        strcat(saida, stack[topo--]);
        strcat(saida, " ");
    }

    return saida;
}

float getValorInFixa(char* StrInFixa) {
    char copia[MAX];
    strcpy(copia, StrInFixa);
    char* posfixa = getFormaPosFixa(copia);
    return getValorPosFixa(posfixa);
}
```

main.c

```
#include "expressao.h"
#include <stdio.h>
#include <string.h>

void testarExpressao(const char* posFixa, const char* inFixaEsperada, float valorEsperado) {
    printf("Pos-fixa: %s\n\n", posFixa);

    char strPosFixa[512];
    strcpy(strPosFixa, posFixa);

    float valor = getValorPosFixa(strPosFixa);
    printf("Valor calculado: %.2f | Valor esperado: %.2f\n", valor, valorEsperado);

    strcpy(strPosFixa, posFixa);
    char* inFixa = getFormaInFixa(strPosFixa);
    printf("Forma infixa: %s | Forma infixa esperada: %s\n\n\n", inFixa, inFixaEsperada);

}

void testarConversaoInFixa(const char* inFixa, const char* posFixaEsperada) {
    printf("Infixa: %s\n", inFixa);

    char strInFixa[512];
    strcpy(strInFixa, inFixa);

    char* posFixa = getFormaPosFixa(strInFixa);
    printf("Forma pos-fixa: %s\nForma pos-fixa esperada: %s\n\n", posFixa, posFixaEsperada);
}

int main() {
    printf("\n---------- Pos-fixada para infixada + calculo da expressao ----------\n\n");
    testarExpressao("3 4 + 5 *", "(3 + 4) * 5", 35);
    testarExpressao("7 2 * 4 +", "7 * 2 + 4", 18);
    testarExpressao("8 5 2 4 + * +", "8 + (5 * (2 + 4))", 38);
    testarExpressao("6 2 / 3 + 4 *", "(6 / 2 + 3) * 4", 24);
    testarExpressao("9 5 2 8 * 4 + * +", "9 + (5 * (2 + 8 * 4))", 109);
    testarExpressao("2 3 + log 5 /", "log(2 + 3) / 5", 0.14f);
    testarExpressao("10 log 3 ^ 2 +", "(log10)^3 + 2", 3);
    testarExpressao("45 60 + 30 cos *", "(45 + 60) * cos(30)", 90.93f);
    testarExpressao("0.5 45 sen 2 ^ +", "sen(45)^2 + 0.5", 1);

    printf("\n---------- Infixada para pos-fixada----------\n\n");
```

```
    testarConversaoInFixa("(3 + 4) * 5", "3 4 + 5 *");
    testarConversaoInFixa("7 * 2 + 4", "7 2 * 4 +");
    testarConversaoInFixa("8 + (5 * (2 + 4))", "8 5 2 4 + * +");
    testarConversaoInFixa("(6 / 2 + 3) * 4", "6 2 / 3 + 4 *");
    testarConversaoInFixa("9 + (5 * (2 + 8 * 4))", "9 5 2 8 * 4 + * +");
    testarConversaoInFixa("log(2 + 3) / 5", "2 3 + log 5 /");
    testarConversaoInFixa("(log10)^3 + 2", "10 log 3 ^ 2 +");
    testarConversaoInFixa("(45 + 60) * cos(30)", "45 60 + 30 cos *");
    testarConversaoInFixa("sen(45) ^2 + 0,5", "0.5 45 sen 2 ^ +");

    return 0;
}
```