Sumário

1. Introdução: 2
2. Implementação: 2
2.1 Estruturas de Dados Utilizadas 2
2.2 Protótipos de Funções e Finalidades 3
2.3 Uso do TAD Pilha 3
3. Testes 5
3.1 Teste n° 01 5
3.2 Teste n° 02 5
3.3 Teste n° 03 6
3.4 Teste n° 04 6
3.5 Teste n° 05 6
3.6 Teste n° 06 7
4. Conclusão 8
5. Referências 8
6. Anexos 8
6.1 expressão.h 8
6.2 expressão.c 9
6.3 main.c 25

1. Introdução:

Este relatório apresenta o desenvolvimento de uma calculadora de expressões matemáticas que realiza conversões entre as notações infixa e pós-fixa, além de calcular o valor de expressões aritméticas. O principal objetivo do trabalho foi aplicar de forma prática o uso do Tipo Abstrato de Dados Pilha (TAD Pilha), mostrando sua eficiência na análise e processamento de expressões matemáticas.

Durante a implementação, o TAD Pilha foi uma ferramenta essencial para lidar com a hierarquia de operações e o correto agrupamento de operandos e operadores. A aplicação construída permite ao usuário interagir com um menu intuitivo, onde é possível converter expressões, calcular seus valores e validar o funcionamento correto das funções.

- Objetivos:
- Converter expressões infixas para pós-fixas;
- Converter expressões pós-fixas para infixas;
- Avaliar numericamente expressões infixas e pós-fixas;
- Suportar operações aritméticas e funções matemáticas (seno, cosseno, tangente, logaritmo e raiz quadrada);
- Tratar erros comuns, como divisão por zero e expressões malformadas.

GitHub:

https://github.com/MViniciusNunes/Calculadora

2. Implementação:

A estrutura do trabalho foi organizada em três arquivos principais:

expressao.h: Contém a definição da estrutura Expressao e os protótipos das funções utilizadas na conversão e avaliação das expressões.
expressao.c: Implementa a lógica completa de conversão entre notações e avaliação numérica, utilizando pilhas auxiliares para controle de precedência e avaliação.
main.c: Responsável pela interface com o usuário, apresentando o menu de opções e capturando as entradas.

2.1 Estruturas de dados Utiizandas

a) PilhaFloat

```
typedef struct {
float itens\[MAX];
int topo;
} PilhaFloat;
```

Descrição: Pilha responsável por armazenar os valores numéricos (float) durante a avaliação de expressões pós-fixas.
Campos:

itens\[MAX]: vetor que armazena os operandos;
topo: controle do topo da pilha.

b) PilhaStr

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

Descrição: Pilha de strings usada na conversão de infixa para pós-fixa, gerenciando operadores e funções.
Campos:

itens\[MAX]\[MAX]: matriz de strings com os tokens de operadores/funções;
topo: controle do topo da pilha.

c) PilhaExpr

```
typedef struct {
char expr\[MAX];
int prioridade;
} ExprComPrioridade;

typedef struct {
ExprComPrioridade itens\[MAX];
int topo;
} PilhaExpr;
```

Descrição: Pilha usada na reconstrução de expressões infixas a partir da pós-fixa, garantindo a correta inserção de parênteses conforme a prioridade dos operadores.
Campos:

expr: subexpressão parcial;
prioridade: nível de prioridade da subexpressão;
topo: controle do topo.

d) Expressao

```
typedef struct {
char posFixa\[512];
char inFixa\[512];
```

float Valor;
} Expressao;

Descrição: Estrutura central que armazena a expressão em ambas as notações e seu valor calculado.
Campos:

posFixa: expressão pós-fixada;
inFixa: expressão infixada;
Valor: resultado numérico.

2.2 Protótipos de Funções e Finalidades

Conversão de Expressões:

char* getFormaPosFixa(char* Str);

Converte expressão infixada em pós-fixada.
Parâmetro: Str — string infixa.
Retorno: string pós-fixa.

char* getFormaInFixa(char* StrPosFixa);

Converte expressão pós-fixada em infixada.
Parâmetro: StrPosFixa — string pós-fixa.
Retorno: string infixa.

Avaliação de Expressões:

float getValorPosFixa(char* StrPosFixa);

Avalia uma expressão pós-fixada.
Parâmetro: StrPosFixa — string pós-fixa.
Retorno: valor numérico calculado.

float getValorInFixa(char* StrInFixa);

Avalia uma expressão infixada (fazendo a conversão interna para pós-fixa).
Parâmetro: StrInFixa — string infixa.
Retorno: valor numérico calculado.

2.3 Uso do TAD Pilha

O TAD Pilha foi o núcleo lógico do trabalho, desempenhando papéis importantes em cada etapa do processamento das expressões:

Conversão Infixa → Pós-fixa:

A pilha de strings (PilhaStr) armazena temporariamente os operadores e funções, garantindo que a ordem de precedência seja respeitada. Quando necessário, operadores são desempilhados e adicionados à saída.

Conversão Pós-fixa → Infixa:

Utiliza-se a PilhaExpr para remontar a expressão infixada de maneira correta. Cada vez que um operador é lido, são retiradas as subexpressões da pilha, inserindo parênteses apenas quando necessário, conforme as prioridades.

Avaliação da Pós-fixa:

A PilhaFloat é responsável por armazenar os operandos. Ao encontrar um operador, são desempilhados dois valores, aplica-se a operação e o resultado volta à pilha.
O comportamento LIFO (último a entrar, primeiro a sair) das pilhas torna-as ideais para controlar a sequência correta de execução das operações matemáticas durante tanto as conversões quanto os cálculos.

3.Testes

Foram realizados testes com expressões de diferentes níveis de complexidade para validar o funcionamento correto da aplicação. A seguir, são apresentados seis testes exemplares:

3.1 Teste n° 01

Expressão Infixa: 3 + 4 * 2
Pós-fixa Gerada: 3 4 2 * +
Resultado Esperado: 11
Processo de Avaliação da Pós-fixa 3 4 2 * +:

Lê 3: Empilha 3.0. -> Pilha: [3.0]
Lê 4: Empilha 4.0. -> Pilha: [3.0, 4.0]
Lê 2: Empilha 2.0. -> Pilha: [3.0, 4.0, 2.0]
Lê *: Desempilha 2.0 e 4.0, calcula 4.0 * 2.0 = 8.0, empilha o resultado. -> Pilha: [3.0, 8.0]
Lê +: Desempilha 8.0 e 3.0, calcula 3.0 + 8.0 = 11.0, empilha o resultado. -> Pilha: [11.0]
Final: Resultado é 11.

3.2 Teste n° 02

Expressão Infixa: (5 + 3) * (2 - 1)
Pós-fixa Gerada: 5 3 + 2 1 - *
Resultado Esperado: 8

Processo de Avaliação da Pós-fixa 5 3 + 2 1 - *:

Lê 5: Empilha 5.0. -> Pilha: [5.0]
Lê 3: Empilha 3.0. -> Pilha: [5.0, 3.0]
Lê +: Calcula 5.0 + 3.0 = 8.0. Empilha 8.0. -> Pilha: [8.0]
Lê 2: Empilha 2.0. -> Pilha: [8.0, 2.0]
Lê 1: Empilha 1.0. -> Pilha: [8.0, 2.0, 1.0]
Lê -: Calcula 2.0 - 1.0 = 1.0. Empilha 1.0. -> Pilha: [8.0, 1.0]
Lê *: Calcula 8.0 * 1.0 = 8.0. Empilha 8.0. -> Pilha: [8.0]
Final: Resultado é 8.

3.3 Teste n° 03

Expressão Infixa: 10 + 2 * 6
Pós-fixa Gerada: 10 2 6 * +
Resultado Esperado: 22
Processo de Avaliação da Pós-fixa 10 2 6 * +:

Lê 10: Empilha 10.0. -> Pilha: [10.0]
Lê 2: Empilha 2.0. -> Pilha: [10.0, 2.0]
Lê 6: Empilha 6.0. -> Pilha: [10.0, 2.0, 6.0]
Lê *: Calcula 2.0 * 6.0 = 12.0. Empilha 12.0. -> Pilha: [10.0, 12.0]
Lê +: Calcula 10.0 + 12.0 = 22.0. Empilha 22.0. -> Pilha: [22.0]
Final: Resultado é 22.

3.4 Teste n° 04

Expressão Infixa: 100 * (2 + 12) / 14
Pós-fixa Gerada: 100 2 12 + * 14 /
Resultado Esperado: 100
Processo de Avaliação da Pós-fixa 100 2 12 + * 14 /:

Lê 100: Empilha 100.0. -> Pilha: [100.0]
Lê 2: Empilha 2.0. -> Pilha: [100.0, 2.0]
Lê 12: Empilha 12.0. -> Pilha: [100.0, 2.0, 12.0]
Lê +: Calcula 2.0 + 12.0 = 14.0. Empilha 14.0. -> Pilha: [100.0, 14.0]
Lê *: Calcula 100.0 * 14.0 = 1400.0. Empilha 1400.0. -> Pilha: [1400.0]
Lê 14: Empilha 14.0. -> Pilha: [1400.0, 14.0]
Lê /: Calcula 1400.0 / 14.0 = 100.0. Empilha 100.0. -> Pilha: [100.0]
Final: Resultado é 100.

3.5 Teste n° 05

Expressão Infixa: sen(30) + cos(60)
Pós-fixa Gerada: 30 sen 60 cos +
Resultado Esperado: 1.50
Processo de Avaliação da Pós-fixa 30 sen 60 cos +:

Lê 30: Empilha 30.0. -> Pilha: [30.0]
Lê sen: Calcula sin(toRadians(30.0)) = 0.5. Empilha 0.5. -> Pilha: [0.5]
Lê 60: Empilha 60.0. -> Pilha: [0.5, 60.0]
Lê cos: Calcula cos(toRadians(60.0)) = 0.5. Empilha 0.5. -> Pilha: [0.5, 0.5]
Lê +: Calcula 0.5 + 0.5 = 1.0. Empilha 1.0. -> Pilha: [1.0]
Final: Resultado é 1.

3.6 Teste n° 06

Expressão Infixa: raiz(16) + log(100)
Pós-fixa Gerada: 16 raiz 100 log +
Resultado Esperado: 6.00
Processo de Avaliação da Pós-fixa 16 raiz 100 log +:

Lê 16: Empilha 16.0. -> Pilha: [16.0]
Lê raiz: Calcula sqrt(16.0) = 4.0. Empilha 4.0. -> Pilha: [4.0]
Lê 100: Empilha 100.0. -> Pilha: [4.0, 100.0]
Lê log: Calcula log10(100.0) = 2.0. Empilha 2.0. -> Pilha: [4.0, 2.0]
Lê +: Calcula 4.0 + 2.0 = 6.0. Empilha 6.0. -> Pilha: [6.0]
Final: Resultado é 6.

4. Conclusão

O trabalho nos permitiu aplicar na prática o uso de pilhas para resolver expressões matemáticas, nos ajudando a entender como as pilhas funcionam, tanto para converter expressões quanto para calcular seus valores.

Dificuldades encontradas:

- Ajustar corretamente a ordem das operações (precedência dos operadores);
- Inserir parênteses de forma adequada nas conversões entre infixa e pós-fixa;
- Implementar o tratamento de erros (divisão por zero, expressões inválidas).

Possíveis melhorias futuras:

- Adicionar mais funções matemáticas no sistema;
- Melhorar a verificação e validação da entrada de dados;
- Criar uma interface que mostre o passo a passo da conversão e do cálculo.

5. Referências

SZWARCFITER, J. L.; MARKENZON, L. *Estruturas de dados e seus algoritmos.* 3. ed. Rio de Janeiro: LTC, 2010.

VILLAS, M. V. et al. *Estruturas de dados: conceitos e técnicas de implementação.* Rio de Janeiro: Campus, 1993.

6. Anexos

https://github.com/MViniciusNunes/Calculadora

6.1 expressao.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 infixa, como 3 * (12 + 4)
```

```
float Valor; // Valor numérico da expressão

} Expressao;


char *getFormaInFixa(char *Str); // Retorna a forma inFixa de Str (posFixa)

char *getFormaPosFixa(char *Str); // Retorna a forma posFixa de Str (inFixa)

float getValorPosFixa(char *StrPosFixa); // Calcula o valor de Str (na forma posFixa)

float getValorInFixa(char *StrInFixa); // Calcula o valor de Str (na forma inFixa)


#endif
```

6.2 expressão.c

```
#include  <stdio.h>

#include <stdlib.h>

#include <string.h>

#include <ctype.h>

#include <math.h>


#define MAX 512


#ifndef M_PI

#define M_PI 3.14159265358979323846

#endif
```

```
// Converte graus para radianos
static double toRadians(double degrees) {
    return degrees * M_PI / 180.0;
}

void toLowerStr(char *str); // <-- Adicione esta linha aqui

//=================================================================
// ESTRUTURAS DE PILHA
//=================================================================

// --- Pilha de Strings (Para conversões de notação) ---
typedef struct {
    char itens[MAX][MAX];
    int topo;
} PilhaStr;

void initPilhaStr(PilhaStr *p) { p->topo = -1; }
int isEmptyStr(PilhaStr *p) { return p->topo == -1; }
void pushStr(PilhaStr *p, char *val) { if (p->topo < MAX - 1) strcpy(p->itens[++(p->topo)], val); }
char* popStr(PilhaStr *p) { return !isEmptyStr(p) ? p->itens[(p->topo)--] : NULL; }
```

```
char* peekStr(PilhaStr *p) { return !isEmptyStr(p) ? p->itens[p->topo] : NULL; }


// --- Pilha de Floats (Para cálculo de valores) ---
typedef struct {
    float itens[MAX];
    int topo;
} PilhaFloat;


void initPilhaFloat(PilhaFloat *p) { p->topo = -1; }
int isEmptyFloat(PilhaFloat *p) { return p->topo == -1; }
void pushFloat(PilhaFloat *p, float val) { if (p->topo < MAX - 1) p->itens[++(p->topo)] = val; }
float popFloat(PilhaFloat *p) { return !isEmptyFloat(p) ? p->itens[(p->topo)--] : 0.0; }


// Estrutura auxiliar para armazenar expressão e prioridade
typedef struct {
    char expr[MAX];
    int prioridade;
} ExprComPrioridade;


// Nova pilha para ExprComPrioridade
typedef struct {
    ExprComPrioridade itens[MAX];
```

```
    int topo;
} PilhaExpr;

void initPilhaExpr(PilhaExpr *p) { p->topo = -1; }
int isEmptyExpr(PilhaExpr *p) { return p->topo == -1; }
void pushExpr(PilhaExpr *p, ExprComPrioridade val) { if (p->topo < MAX - 1) p->itens[++(p->topo)] = val; }
ExprComPrioridade popExpr(PilhaExpr *p) { return p->itens[(p->topo)--]; }

//=====================================================================
// FUNÇÕES AUXILIARES
//=====================================================================

int prioridade(char *op) {
    char opCopia[MAX];
    strcpy(opCopia, op);
    toLowerStr(opCopia);

    if (strcmp(opCopia, "sen") == 0 || strcmp(opCopia, "cos") == 0 || strcmp(opCopia, "tg") == 0 ||
        strcmp(opCopia, "log") == 0 || strcmp(opCopia, "raiz") == 0) return 4;
    if (strcmp(opCopia, "^") == 0) return 3;
    if (strcmp(opCopia, "*") == 0 || strcmp(opCopia, "/") == 0 || strcmp(opCopia, "%") == 0) return 2;
    if (strcmp(opCopia, "+") == 0 || strcmp(opCopia, "-") == 0) return 1;
```

```
    return 0; // Para parênteses
}

int ehOperador(char *token) {
    return strcmp(token, "+") == 0 || strcmp(token, "-") == 0 ||
           strcmp(token, "*") == 0 || strcmp(token, "/") == 0 ||
           strcmp(token, "%") == 0 || strcmp(token, "^") == 0 ||
           strcmp(token, "log") == 0 || strcmp(token, "sen") == 0 ||
           strcmp(token, "cos") == 0 || strcmp(token, "tg") == 0 ||
           strcmp(token, "raiz") == 0;
}

void toLowerStr(char *str) {
    for (int i = 0; str[i]; i++) {
        str[i] = tolower((unsigned char)str[i]);
    }
}

//================================================================
// FUNÇÕES PRINCIPAIS COM VALIDAÇÃO COMPLETA
//================================================================
```

```
/**
 * Converte Infixa para Pós-fixa, com validação de erros.
 */
char *getFormaPosFixa(char *Str) {
    static char saida[MAX] = "";
    PilhaStr pilha;
    initPilhaStr(&pilha);
    saida[0] = '\0';

    char token[MAX];
    int i = 0;
    while (Str[i] != '\0') {
        if (isspace(Str[i])) {
            i++;
            continue;
        }

        if (isdigit(Str[i]) || Str[i] == '.') {
            int j = 0;
            while (isdigit(Str[i]) || Str[i] == '.') {
                token[j++] = Str[i++];
            }
```

```
            token[j] = '\0';
            strcat(saida, token);
            strcat(saida, " ");
        } else if (isalpha(Str[i])) {
            int j = 0;
            while (isalpha(Str[i])) {
                token[j++] = Str[i++];
            }
            token[j] = '\0';
            char tokenCopia[MAX];
            strcpy(tokenCopia, token);
            toLowerStr(tokenCopia);
            if(strcmp(tokenCopia, "sen") != 0 && strcmp(tokenCopia, "cos") != 0 && strcmp(tokenCopia, "tg") !=
0 && strcmp(tokenCopia, "log") != 0 && strcmp(tokenCopia, "raiz") != 0) {
                printf("ERRO: Funcao desconhecida: '%s'\n", token);
                return NULL;
            }
            pushStr(&pilha, tokenCopia);
        } else if (Str[i] == '(') {
            token[0] = '('; token[1] = '\0';
            pushStr(&pilha, token);
            i++;
```

```
    } else if (Str[i] == ')') {
        while (!isEmptyStr(&pilha) && strcmp(peekStr(&pilha), "(") != 0) {
            strcat(saida, popStr(&pilha));
            strcat(saida, " ");
        }
        if (isEmptyStr(&pilha)) {
            printf("ERRO: Parentese de fechamento ')' sem um par de abertura correspondente.\n");
            return NULL;
        }
        popStr(&pilha); // remove '('
        i++;
    } else { // Operador
        token[0] = Str[i]; token[1] = '\0';
        if (!ehOperador(token)) {
            printf("ERRO: Operador ou caractere invalido: '%s'\n", token);
            return NULL;
        }
        while (!isEmptyStr(&pilha) && strcmp(peekStr(&pilha), "(") != 0 && prioridade(peekStr(&pilha)) >=
prioridade(token)) {
            strcat(saida, popStr(&pilha));
            strcat(saida, " ");
        }
```

```
            pushStr(&pilha, token);
            i++;
        }
    }

    while (!isEmptyStr(&pilha)) {
        char *op = popStr(&pilha);
        if (strcmp(op, "(") == 0) {
            printf("ERRO: Parentese de abertura '(' sem um par de fechamento correspondente.\n");
            return NULL;
        }
        strcat(saida, op);
        strcat(saida, " ");
    }

    int len = strlen(saida);
    if (len > 0 && saida[len - 1] == ' ')
        saida[len - 1] = '\0';

    return saida;
}
```

```
/**
 * Converte Pós-fixa para Infixa, com validação completa de erros.
 */
char *getFormaInFixa(char *StrPosFixa) {
    static char inFixaResult[MAX];
    PilhaExpr pilha;
    initPilhaExpr(&pilha);

    char strCpy[MAX];
    strcpy(strCpy, StrPosFixa);
    char *token = strtok(strCpy, " ");

    while (token != NULL) {
        ExprComPrioridade novo;
        char tokenCopia[MAX];
        strcpy(tokenCopia, token);
        toLowerStr(tokenCopia);

        if (isdigit(token[0]) || (token[0] == '-' && isdigit(token[1]))) {
            snprintf(novo.expr, MAX, "%s", token);
            novo.prioridade = 10; // prioridade máxima para operandos
            pushExpr(&pilha, novo);
```

```
    } else if (ehOperador(tokenCopia)) {
        int prio = prioridade(tokenCopia);
        if (strcmp(tokenCopia, "sen") == 0 || strcmp(tokenCopia, "cos") == 0 || strcmp(tokenCopia, "tg") ==
0 ||
            strcmp(tokenCopia, "log") == 0 || strcmp(tokenCopia, "raiz") == 0) {
            if (isEmptyExpr(&pilha)) {
                printf("ERRO: Expressao pos-fixa malformada. Operador '%s' sem operandos.\n", token);
                return NULL;
            }
            ExprComPrioridade op = popExpr(&pilha);
            snprintf(novo.expr, MAX, "%s(%s)", tokenCopia, op.expr);
            novo.prioridade = 4; // prioridade das funções
            pushExpr(&pilha, novo);
        } else {
            if (pilha.topo < 1) {
                printf("ERRO: Expressao pos-fixa malformada. Operador '%s' sem operandos suficientes.\n",
token);
                return NULL;
            }
            ExprComPrioridade op2 = popExpr(&pilha);
            ExprComPrioridade op1 = popExpr(&pilha);

            // Adiciona parênteses se a prioridade do operando for menor que a do operador atual
```

```
            char esq[MAX], dir[MAX];
            if (op1.prioridade < prio)
                snprintf(esq, MAX, "(%s)", op1.expr);
            else
                snprintf(esq, MAX, "%s", op1.expr);

            if (op2.prioridade < prio || (prio == 3 && op2.prioridade == prio)) // para associatividade à direita
do ^
                snprintf(dir, MAX, "(%s)", op2.expr);
            else
                snprintf(dir, MAX, "%s", op2.expr);

            snprintf(novo.expr, MAX, "%s %s %s", esq, tokenCopia, dir);
            novo.prioridade = prio;
            pushExpr(&pilha, novo);
        }
    } else {
        printf("ERRO: Token invalido na expressao pos-fixa: '%s'\n", token);
        return NULL;
    }
    token = strtok(NULL, " ");
  }
```

```
    if (pilha.topo != 0) {
        printf("ERRO: Expressao pos-fixa malformada. Sobraram operandos.\n");
        return NULL;
    }
    strcpy(inFixaResult, pilha.itens[0].expr);
    return inFixaResult;
}

/**
 * Calcula o valor de uma expressão Pós-fixa, com tratamento de erros.
 */
float getValorPosFixa(char *StrPosFixa) {
    PilhaFloat pilha;
    initPilhaFloat(&pilha);

    char strCpy[MAX];
    strcpy(strCpy, StrPosFixa);
    char *token = strtok(strCpy, " ");

    while (token != NULL) {
        if (isdigit(token[0]) || (token[0] == '-' && isdigit(token[1]))) {
```

```
            pushFloat(&pilha, atof(token));
        } else { // Se não for número, é um operador ou função
            char tokenLower[MAX];
            strcpy(tokenLower, token);
            toLowerStr(tokenLower);

            // Funções unárias
            if (strcmp(tokenLower, "sen") == 0 || strcmp(tokenLower, "cos") == 0 || strcmp(tokenLower, "tg") ==
0 ||
                strcmp(tokenLower, "log") == 0 || strcmp(tokenLower, "raiz") == 0) {

                if (isEmptyFloat(&pilha)) {
                    printf("ERRO: Expressao malformada. Operador '%s' sem operandos suficientes.\n", token);
                    return NAN;
                }
                float operando = popFloat(&pilha);

                if (strcmp(tokenLower, "sen") == 0) {
                    pushFloat(&pilha, sin(operando * M_PI / 180.0)); // Converte para radianos
                } else if (strcmp(tokenLower, "cos") == 0) {
                    pushFloat(&pilha, cos(operando * M_PI / 180.0));
                } else if (strcmp(tokenLower, "tg") == 0) {
```

```
        pushFloat(&pilha, tan(operando * M_PI / 180.0));
    } else if (strcmp(tokenLower, "log") == 0) {
        pushFloat(&pilha, log10(operando));
    } else if (strcmp(tokenLower, "raiz") == 0) {
        pushFloat(&pilha, sqrt(operando));
    }
} else { // Operadores binários
    if (pilha.topo < 1) {
        printf("ERRO: Expressao malformada. Operador '%s' sem operandos suficientes.\n", token);
        return NAN;
    }
    float op2 = popFloat(&pilha);
    float op1 = popFloat(&pilha);

    if (strcmp(tokenLower, "+") == 0) pushFloat(&pilha, op1 + op2);
    else if (strcmp(tokenLower, "-") == 0) pushFloat(&pilha, op1 - op2);
    else if (strcmp(tokenLower, "*") == 0) pushFloat(&pilha, op1 * op2);
    else if (strcmp(tokenLower, "/") == 0) {
        if (op2 == 0) {
            printf("ERRO: Divisao por zero.\n");
            return NAN;
        }
```

```
            pushFloat(&pilha, op1 / op2);
        }
        else if (strcmp(tokenLower, "%") == 0) {
            if ((int)op2 == 0) {
                printf("ERRO: Modulo por zero.\n");
                return NAN;
            }
            pushFloat(&pilha, fmod(op1, op2));
        }
        else if (strcmp(tokenLower, "^") == 0) pushFloat(&pilha, pow(op1, op2));
        else {
            printf("ERRO: Funcao ou operador desconhecido: '%s'\n", token);
            return NAN;
        }
      }
    }
    token = strtok(NULL, " ");
  }
  return popFloat(&pilha);
}

/**
```

```
 * Calcula o valor de uma expressão Infixa, com tratamento de erros.
 */
float getValorInFixa(char *StrInFixa) {
    char *posFixa = getFormaPosFixa(StrInFixa);
    // Se a conversão falhar (retornar NULL), propaga o erro como NAN
    if (posFixa == NULL) {
        return NAN;
    }
    return getValorPosFixa(posFixa);
}
```

6.3 main.c

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

 int main() {
    int opcao = -1; // Inicializa com um valor inválido
    char entrada[512]; // Buffer temporário para a entrada do usuário
    Expressao minhaExpressao; // Declara uma variável do tipo Expressao
```

```
while (1) {

    printf("\n==== MENU ====\n");

    printf("1 - Traduzir expressao infixada -> pos-fixada\n");

    printf("2 - Traduzir expressao pos-fixada -> infixada\n");

    printf("3 - Avaliar expressao infixada\n");

    printf("4 - Avaliar expressao pos-fixada\n");

    printf("0 - Sair\n");

    printf("Escolha uma opcao: ");


    // --- LÓGICA DE VALIDAÇÃO DE ENTRADA DO MENU ---

    char buffer[512];

    char char_extra;

    fgets(buffer, sizeof(buffer), stdin);


    if (sscanf(buffer, "%d %c", &opcao, &char_extra) != 1) {

        opcao = -1; // Define como inválida se houver lixo ou se não for um número.

    }

    //-------------------------------------------------------


    switch (opcao) {

        case 1:

            printf("Digite a expressao infixada:\n> ");
```

```
fgets(entrada, sizeof(entrada), stdin);

entrada[strcspn(entrada, "\n")] = '\0';


// Substitui vírgula por ponto na entrada

for (int i = 0; entrada[i]; i++) {

   if (entrada[i] == ',') entrada[i] = '.';

}


// Copia a entrada infixa para a struct

strcpy(minhaExpressao.inFixa, entrada);


// Chama a função para obter a forma pós-fixa

char *posFixaResult = getFormaPosFixa(minhaExpressao.inFixa);


if (posFixaResult == NULL) {

   printf(">> ERRO: Nao foi possivel traduzir a expressao.\n");

} else {

   strcpy(minhaExpressao.posFixa, posFixaResult); // Guarda na struct

   printf("Expressao pos-fixada: %s\n", minhaExpressao.posFixa);

}

break;
```

```
case 2:
    printf("Digite a expressao pos-fixada:\n> ");
    fgets(entrada, sizeof(entrada), stdin);
    entrada[strcspn(entrada, "\n")] = '\0';

    // Copia a entrada pós-fixa para a struct
    strcpy(minhaExpressao.posFixa, entrada);

    // Chama a função para obter a forma infixa
    char *inFixaResult = getFormaInFixa(minhaExpressao.posFixa);
    if (inFixaResult == NULL) {
        printf(">> ERRO: Nao foi possivel traduzir a expressao.\n");
    } else {
        strcpy(minhaExpressao.inFixa, inFixaResult); // Guarda na struct
        printf("Expressao infixada: %s\n", minhaExpressao.inFixa);
    }
    break;

case 3:
    printf("Digite a expressao infixada:\n> ");
    fgets(entrada, sizeof(entrada), stdin);
    entrada[strcspn(entrada, "\n")] = '\0';
```

```
// Substitui vírgula por ponto na entrada
for (int i = 0; entrada[i]; i++) {
    if (entrada[i] == ',') entrada[i] = '.';
}

// Copia a entrada infixa para a struct
strcpy(minhaExpressao.inFixa, entrada);

// Chama a função para calcular o valor da expressão infixa
minhaExpressao.Valor = getValorInFixa(minhaExpressao.inFixa); // Guarda na struct

if (isnan(minhaExpressao.Valor) || isinf(minhaExpressao.Valor)) {
    printf(">> ERRO: Nao foi possivel calcular o valor da expressao.\n");
} else {
    if (fabs(minhaExpressao.Valor - (int)minhaExpressao.Valor) < 0.00001) {
        printf("Resultado: %d\n", (int)minhaExpressao.Valor);
    } else {
        printf("Resultado: %.2f\n", minhaExpressao.Valor);
    }
}
break;
```

```
case 4:
    printf("Digite a expressao pos-fixada:\n> ");
    fgets(entrada, sizeof(entrada), stdin);
    entrada[strcspn(entrada, "\n")] = '\0';

    // Copia a entrada pós-fixa para a struct
    strcpy(minhaExpressao.posFixa, entrada);

    // Chama a função para calcular o valor da expressão pós-fixa
    minhaExpressao.Valor = getValorPosFixa(minhaExpressao.posFixa); // Guarda na struct

    if (isnan(minhaExpressao.Valor) || isinf(minhaExpressao.Valor)) {
        printf(">> ERRO: Nao foi possivel calcular o valor da expressao.\n");
    } else {
        if (fabs(minhaExpressao.Valor - (int)minhaExpressao.Valor) < 0.00001) {
            printf("Resultado: %d\n", (int)minhaExpressao.Valor);
        } else {
            printf("Resultado: %.2f\n", minhaExpressao.Valor);
        }
    }
    break;
```

```
        case 0:
            printf("Encerrando o programa.\n");
            return 0;

        default:
            printf("Opcao invalida. Tente novamente.\n");
      }
   }

   return 0;
}
```