Sumário

1. Introdução: 2
2. Implementação: 3
2.1 Uso do TAD Pilha 8
3. Testes 10
3.1 Teste n° 01 11
3.2 Teste n° 02 13
3.3 Teste n° 03 15
3.4 Teste n° 04 17
3.5 Teste n° 05 25
3.6 Teste n° 06 27
4. Conclusão 29
Referências 30
Anexos 31
calculadora.h 32
calculadora.c #include <stdio.h> 33
main.c 41

1. Introdução:

Este trabalho aborda o desenvolvimento de um avaliador de expressões numéricas em linguagem C. O objetivo principal do projeto é aplicar os **conceitos** de **Tipos Abstratos de Dados (TAD), com foco especial na estrutura de dados Pilha**, para criar uma ferramenta capaz de resolver problemas computacionais práticos.

O problema central da questão envolve a interpretação de expressões (Notações) matemáticas. Humanos utilizam a **notação infixa** (ex: 5 + 3 * 2), que posiciona os operadores entre os operandos e depende de regras de **ordem de prioridade** (precedência) e parênteses para evitar ambiguidades. Computadores, por outro lado, processam mais eficientemente a **notação pós-fixa**, ou Polonesa Reversa (ex: 5 3 2 * +), onde o operador sempre aparece depois de seus operandos. Esta notação é inerentemente livre de ambiguidades e não necessita de parênteses. Portanto, o desafio consiste em criar uma solução que possa **converter** uma expressão da forma humana (infixa) para a forma computacional (pós-fixa), respeitando a hierarquia dos operadores, e, em seguida, **calcular** o resultado final a partir dessa nova representação.

O programa desenvolvido é capaz de realizar(basicamente) as seguintes operações:

- Traduzir expressões da notação Pós-fixa de volta para a notação Infixa.
- Traduzir expressões matemáticas da notação Infixa para a Pós-fixa.
- Calcular o resultado numérico de expressões em qualquer uma das duas notações.
- Suportar os operadores matemáticos básicos (+, -, *, /, %, ^) e funções unárias como raiz quadrada (raiz), seno (sen), cosseno (cos), tangente (tg) e logaritmo na base 10 (log).

GitHub:

https://github.com/octavios-sign/Avaliacao-de-expressao-numerica

2. Implementação:

A solução do problema foi construída sobre dois algoritmos fundamentais da ciência da computação, ambos dependentes do uso de pilhas. A estratégia foi primeiro traduzir a expressão para um formato ideal para máquinas e, depois, executar o cálculo. Foram consultadas também resoluções em outras linguagens de Programação, como Python.

O primeiro passo utiliza o algoritmo Shunting-Yard para converter a expressão da notação infixa para a pós-fixa. Este algoritmo funciona como uma estação de manobra de trens (daí o nome "Shunting-Yard"), usando uma pilha para reorganizar os operadores (+, *, ^, etc.) com base em suas prioridades. Ao ler a expressão de entrada, números são enviados diretamente para uma fila de saída, enquanto operadores são temporariamente colocados em uma pilha. Um operador só é empilhado após garantir que nenhum outro operador de maior prioridade esteja no topo da pilha; caso contrário, o operador do topo é movido para a saída primeiro. Parênteses servem como regras especiais para forçar uma ordem de empilhamento e desempilhamento, garantindo que sub-expressões sejam tratadas corretamente.

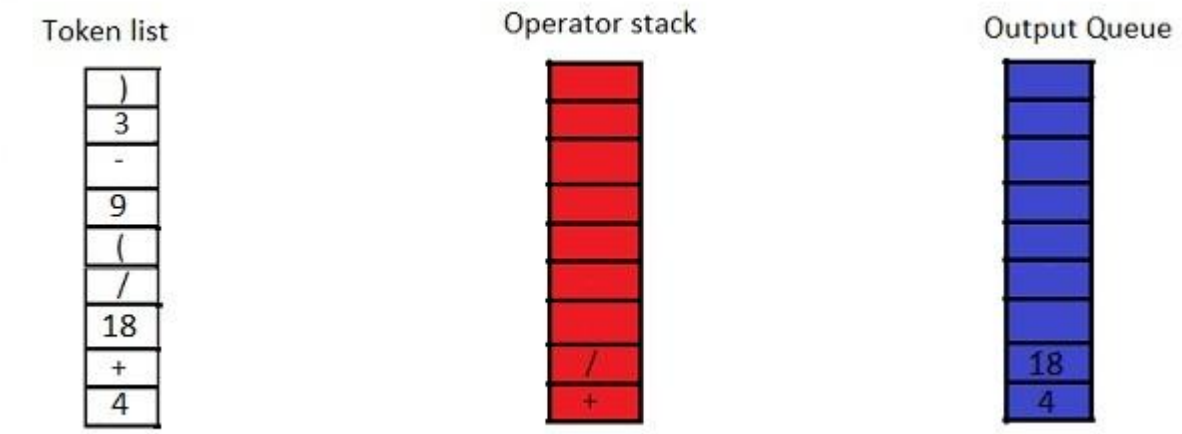

Figura 1 - Shunting-yard algorithm

Com a expressão já no formato pós-fixo, o segundo passo é a avaliação da expressão, um processo muito mais linear. Utilizando uma segunda pilha, desta vez para operandos (números), o algoritmo percorre a expressão pós-fixa. Se o item lido for um número, ele é simplesmente empilhado. Se for um operador, o algoritmo desempilha a quantidade necessária de operandos (dois para operadores binários como + e *, um

para funções unárias como log), realiza o cálculo e empilha o resultado de volta. Ao final da expressão, o único valor restante na pilha é a resposta final.

Para implementar essa lógica, o código foi estruturado com as seguintes ferramentas:

- Estruturas de Dados: Foram definidas duas implementações do TAD Pilha:
- **StackStr**: Uma pilha de strings (vetor de char), projetada para armazenar os tokens de operadores e funções durante a conversão pelo Shunting-Yard.
- **StackFloat**: Uma pilha de float, otimizada para armazenar os operandos numéricos durante a fase de cálculo.
- Funções Principais e Auxiliares: A biblioteca calculadora.c contém, além das funções principais (**getFormaPosFixa**, **getFormaInFixa**, **getValorPosFixa**, **getValorInFixa**), um conjunto de funções auxiliares que tornam o código mais legível e "modular". As Funções **isOperator()**, **isFunction()** e **precedence()** abstraem as verificações de tokens, limpando a lógica dos algoritmos principais. A função **tokenize()** é particularmente importante, pois é responsável por quebrar a string de entrada em uma lista de "tokens" significativos (números, operadores, parênteses), que é o formato de entrada esperado pelos algoritmos.

Bibliotecas Utilizadas:

Para a construção do programa, foram incluídas as seguintes bibliotecas padrão da linguagem C, cada uma com um propósito específico:

- **<stdio.h>** : Para interação com o usuário. Foi utilizada para funções como printf (para exibir o menu, os resultados e as explicações) e fgets/scanf (para ler a expressão digitada pelo usuário).
- **<stdlib.h>** : Fornece funções de propósito geral. Seu uso principal neste projeto foi a função atof, que converte uma string contendo um número para seu valor correspondente em float, sendo crucial na fase de cálculo.
- **<string.h>** : Fundamental para toda a manipulação de texto. Funções como strcpy e strcat foram usadas para construir as strings de saída nas conversões, e strcmp foi usada extensivamente para comparar os tokens lidos com os operadores e funções conhecidos.
- **<ctype.h>** : Contém funções para classificar e converter caracteres. As funções isdigit e isalpha foram importantes na rotina de tokenize para separar a expressão de entrada em números, operadores e funções. A função tolower foi usada para padronizar a entrada do usuário.
- **<math.h>** : Para a execução dos cálculos matemáticos. Forneceu as funções **pow** (potência), **sqrt** (raiz quadrada), **log10** (logaritmo na base 10), **sin**, **cos**, **tan** (funções trigonométricas) e **fmod** (resto

da divisão para float). **Sua utilização requer a linkagem com a flag -lm durante a compilação.**

Atendimento aos Requisitos do Projeto:

A seguir, detalha-se como cada requisito funcional do trabalho foi atendido no código-fonte.

- **A) Tradução da notação Infixa para Pós-fixa**

  Este requisito é atendido pela função **getFormaPosFixa()**. Ela implementa o algoritmo Shunting-Yard, que utiliza uma pilha para reorganizar os operadores de acordo com sua precedência e associatividade. **O trecho de código central que gerencia essa lógica de prioridades é o while dentro do laço principal, que decide se um operador do topo da pilha deve ser movido para a saída antes de empilhar um novo.**

calculadora.c:

```
// Dentro da função getFormaPosFixa, ao encontrar um operador
while (!emptyStr(&opStack) && strcmp(peekStr(&opStack), "(") != 0 &&
     (precedence(peekStr(&opStack)) > precedence(tokens[i]) ||
      (precedence(peekStr(&opStack)) == precedence(tokens[i]) && isLeftAssociative(tokens[i])))) {
   strcpy(output[outSize++], popStr(&opStack));
}
pushStr(&opStack, tokens[i]);
```

- **B) Tradução da notação Pós-fixa para Infixa**

A conversão inversa é realizada pela função **getFormaInFixa()**. Ela utiliza uma pilha de strings para construir a expressão infixa. Números são empilhados diretamente. Ao encontrar um operador, a função desempilha duas sub-expressões, e as une com o operador no meio e envolve o resultado em parênteses para garantir que a ordem das operações seja preservada visualmente.

calculadora.c:

```
// Dentro da função getFormaInFixa, ao encontrar um operador
char b[128], a[128];
strcpy(b, popStr(&s));
strcpy(a, popStr(&s));
sprintf(temp, "(%s %s %s)", a, tokens[i], b);
pushStr(&s, temp);
```

- **C) Suporte aos operadores matemáticos básicos**

O programa suporta os operadores +, -, *, /, % e também ^ (potência). O reconhecimento desses operadores é feito pela função auxiliar **isOperator()**, e a lógica de cálculo para cada um está localizada na função **getValorPosFixa()**, onde dois operandos são desempilhados e a operação correspondente é aplicada.

calculadora.c:

```
if (strcmp(tokens[i], "+") == 0) r = a + b;
else if (strcmp(tokens[i], "-") == 0) r = a - b;
else if (strcmp(tokens[i], "*") == 0) r = a * b;
else if (strcmp(tokens[i], "/") == 0) { if (b == 0) return NAN; r = a / b; }
else if (strcmp(tokens[i], "%") == 0) { if (b == 0) return NAN; r = fmod(a, b); }
else if (strcmp(tokens[i], "^") == 0) r = pow(a, b);
```

- **D) e F) Suporte a funções matemáticas de um operando**

As funções raiz, sen, cos, tg e log foram implementadas para operar sobre um único operando, conforme o requisito (F). A função **getValorPosFixa()** verifica se o token é uma função com **isFunction()** e, em caso afirmativo, desempilha apenas um valor da pilha de operandos para realizar o cálculo.

calculadora.c:

```
else if (isFunction(tokens[i])) {
   if (sizeFloat(&s) < 1) return NAN; // Garante que há um operando
   float a = popFloat(&s);          // Desempilha SOMENTE UM operando
   float r = 0.0;
   // ...
```

- **E) Tratamento de ângulos em graus**

Para que as funções trigonométricas aceitem ângulos em graus, foi necessário realizar a conversão para radianos, que é o formato esperado pelas funções da biblioteca **<math.h>**. Essa conversão é feita no momento do cálculo, multiplicando o valor em **graus** pela **constante fracpi180**.

calculadora.c:

```
if (strcmp(tokens[i], "sen") == 0) r = sin(a * M_PI / 180.0);
else if (strcmp(tokens[i], "cos") == 0) r = cos(a * M_PI / 180.0);
else if (strcmp(tokens[i], "tg") == 0) r = tan(a * M_PI / 180.0);
```

- **G) Avaliação de expressões da tabela**

A combinação de todas as funcionalidades implementadas (tokenização, conversão e cálculo) permite que o programa avalie corretamente as expressões propostas na tabela do enunciado.

2.1 Uso do TAD Pilha

O Tipo Abstrato de Dados Pilha (LIFO - Last-In, First-Out) é a peça central de todo o sistema. O nosso projeto utiliza duas pilhas com propósitos distintos: uma para os operadores durante a conversão e outra para os operandos durante o cálculo.

Na conversão de Infixa para Pós-fixa, a pilha atua como uma "área de espera" para os operadores, garantindo que a ordem de precedência seja respeitada. O diagrama abaixo ilustra visualmente o funcionamento do algoritmo Shunting-Yard para a expressão 3 + 5 * 2.

Diagrama 1: Processo de Conversão (Shunting-Yard)

Passo	Token	Ação / Regra Aplicada	Pilha de Operadores	Saída (Pós-fixa)
1	**3**	**É um número. Vai direto para a fila de saída.**	**[(vazia)]**	**3**
2	**+**	**É um operador. A pilha está vazia, então ele é empilhado.**	**[+]**	**3**
3	**5**	**É um número. Vai direto para a fila de saída.**	**[+]**	**3 5**
4		***É um operador. Sua prioridade é maior que a do '+' no topo da pilha, então ele é empilhado.***	***[+, *]***	***35***
5	***2***	***É um número. Vai direto para a fila de saída.***	***[+, *]***	***3 5 2***
6	***(Fim)***	***Fim da expressão. Desempilha todos os operadores restantes (*" e depois '+') para a saída.**	**[(vazia)]**	**3 5 2 * +**

D**e forma análoga, a conversão de Pós-fixa para Infixa também depende de uma pilha**. Nesse caso, a pilha armazena sub-expressões em formato de texto. Ao ler um número, ele é empilhado. Ao encontrar um operador, duas sub-expressões são desempilhadas, unidas pelo operador e envolvidas em parênteses (ex: (A + B)), e este novo texto é empilhado de volta. Esse processo reconstrói a expressão na notação infixa, garantindo a ordem correta das operações.

Finalmente, na avaliação da expressão Pós-fixa, a pilha funciona como a memória de cálculo. Ela guarda os números até que uma operação precise ser realizada. O diagrama a seguir mostra o cálculo da expressão 3 5 2 * +, gerada no passo anterior.

Diagrama 2: Processo de Avaliação para calcular resultados de uma Expressão Pós-fixa

Ação	Pilha de Operandos
Lendo token '3'. Empilha.	[3]
Lendo token '5'. Empilha.	[3, 5]
Lendo token '2'. Empilha.	[3, 5, 2]
Lendo token '*'. Desempilha 2 e 5, calcula 5 * 2 = 10, empilha 10.	[3, 10]
Lendo token '+'. Desempilha 10 e 3, calcula 3 + 10 = 13, empilha 13.	[13]
Fim da expressão. O resultado final é 13.	[13]

3. Testes

Os testes foram selecionados para cobrir diferentes cenários, desde operações simples até expressões aninhadas com funções matemáticas. Durante este processo, algumas dificuldades foram encontradas, principalmente relacionadas à conversão entre notações e à detecção automática do tipo de expressão.

Para oferecer uma experiência de usuário mais fluida, o programa tenta identificar se a expressão é infixa ou pós-fixa. A heurística para isso é simples: a ausência de parênteses e a presença de um padrão numérico inicial indicam uma expressão pós-fixa. Embora funcional, essa abordagem simplificada é um ponto de atenção, como observado no Teste 2 da Tabela(Documento dos Anexos), onde uma expressão infixa sem

parênteses (7 * 2 + 4) poderia ser confundida com uma pós-fixa; foram feitos ajustes na heurística para tratar esses casos.

Uma dificuldade notável surgiu ao comparar os resultados da conversão do programa com a tabela fornecida no enunciado, especificamente nos Testes 5 e 9. A análise revelou que as notações infixa e pós-fixa na tabela eram inconsistentes entre si. Por exemplo, para o Teste 5, a expressão infixa 9 + (5 * (2 + 8 * 4)) (resultado 179) não corresponde à expressão pós-fixa 9 5 2 8 * 4 + * + (resultado 109). De forma similar, a conversão padrão de sen(45)^2+0.5 (Teste 9) resulta em 45 sen 2 ^ 0.5 +, enquanto a tabela apresentava uma ordem diferente. Foi feita uma decisão de projeto de manter a implementação fiel aos algoritmos padrões (Shunting-Yard), garantindo um comportamento consistente e correto para todas as expressões, em vez de criar exceções para corresponder a conversões inconsistentes. A regra que o programa utiliza para identificar uma expressão como pós-fixa é verificar a "ausência de parênteses e a presença de um padrão numérico inicial (dois números seguidos identificados nos Tokens).

3.1 Teste n° 01

==========

Entrada - ANÁLISE DA EXPRESSAO: (3 + 4) * 5

==========

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---

Expressão identificada como: INFIXA.

Convertendo para a forma Pos-fixa para o cálculo.

- Original (Infixa): (3 + 4) * 5

- Convertida (Pos-fixa): 3 4 + 5 *

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "3"

-> É um número. Empilha 3.

Pilha atual: [3.00]

Lendo token: "4"

-> É um número. Empilha 4.

Pilha atual: [3.00 4.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 3.00 (operando 'a').

- Calcula: 3.00 + 4.00 = 7.00

- Empilha o resultado 7.00.

Pilha atual: [7.00]

Lendo token: "5"

-> É um número. Empilha 5.

Pilha atual: [7.00 5.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 5.00 (operando 'b') e 7.00 (operando 'a').

- Calcula: 7.00 * 5.00 = 35.00

- Empilha o resultado 35.00.

Pilha atual: [35.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 35

3.2 Teste n° 02

```
============================================================
      Entrada  -  ANÁLISE DA EXPRESSAO: 7 2 * 4 +
============================================================

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---
Expressão identificada como: POS-FIXA.
Convertendo para a forma infixa para visualização.
 - Original (Pos-fixa): 7 2 * 4 +
 - Convertida (Infixa): (7 * 2) + 4

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---
O cálculo e feito lendo cada item (token) da expressão pos-fixa.
Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "7"
 -> É um número. Empilha 7.
    Pilha atual: [ 7.00 ]

Lendo token: "2"
 -> É um número. Empilha 2.
    Pilha atual: [ 7.00 2.00 ]

Lendo token: "*"
 -> É um operador. Requer 2 operandos.
    - Desempilha 2.00 (operando 'b') e 7.00 (operando 'a').
    - Calcula: 7.00 * 2.00 = 14.00
    - Empilha o resultado 14.00.
    Pilha atual: [ 14.00 ]
```

Lendo token: "4"

-> É um número. Empilha 4.

Pilha atual: [14.00 4.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 14.00 (operando 'a').

- Calcula: 14.00 + 4.00 = 18.00

- Empilha o resultado 18.00.

Pilha atual: [18.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 18

Obs: aqui houve nosso primeiro problema, ao realizar o teste com a forma Infixa 7 * 2 + 4, ele reconhecia como Pos-fixa, pois antes de conseguirmos nosso resultado, o código estava com ajuste para identificar pos-fixas com ausência de parentes, e infixas com presença. No entanto, isso não é o que define uma expressão Infixa, foram realizados os devidos ajustes.

3.3 Teste n° 03

==========

Entrada - ANÁLISE DA EXPRESSAO: 8 5 2 4 + * +

==========

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---

Expressão identificada como: POS-FIXA.

Convertendo para a forma infixa para visualização.

- Original (Pos-fixa): 8 5 2 4 + * +

- Convertida (Infixa): 8 + (5 * (2 + 4))

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "8"

-> É um número. Empilha 8.

Pilha atual: [8.00]

Lendo token: "5"

-> É um número. Empilha 5.

Pilha atual: [8.00 5.00]

Lendo token: "2"

-> É um número. Empilha 2.

Pilha atual: [8.00 5.00 2.00]

Lendo token: "4"

-> É um número. Empilha 4.

Pilha atual: [8.00 5.00 2.00 4.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 2.00 (operando 'a').

- Calcula: 2.00 + 4.00 = 6.00

- Empilha o resultado 6.00.

Pilha atual: [8.00 5.00 6.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 6.00 (operando 'b') e 5.00 (operando 'a').

- Calcula: 5.00 * 6.00 = 30.00

- Empilha o resultado 30.00.

Pilha atual: [8.00 30.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 30.00 (operando 'b') e 8.00 (operando 'a').

- Calcula: 8.00 + 30.00 = 38.00

- Empilha o resultado 38.00.

Pilha atual: [38.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 38

3.4 Teste n° 04

==========

Entrada - ANÁLISE DA EXPRESSAO: 9 5 2 8 * 4 + * +

==========

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---

Expressão identificada como: POS-FIXA.

Convertendo para a forma Infixa para visualizacao.

- Original (Pos-fixa): 9 5 2 8 * 4 + * +

- Convertida (Infixa): 9 + (5 * ((2 * 8) + 4))

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "9"

-> É um número. Empilha 9.

Pilha atual: [9.00]

Lendo token: "5"

-> É um número. Empilha 5.

Pilha atual: [9.00 5.00]

Lendo token: "2"

-> É um número. Empilha 2.

Pilha atual: [9.00 5.00 2.00]

Lendo token: "8"

-> É um número. Empilha 8.

Pilha atual: [9.00 5.00 2.00 8.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 8.00 (operando 'b') e 2.00 (operando 'a').

- Calcula: 2.00 * 8.00 = 16.00

- Empilha o resultado 16.00.

Pilha atual: [9.00 5.00 16.00]

Lendo token: "4"

-> É um número. Empilha 4.

Pilha atual: [9.00 5.00 16.00 4.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 16.00 (operando 'a').

- Calcula: 16.00 + 4.00 = 20.00

- Empilha o resultado 20.00.

Pilha atual: [9.00 5.00 20.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 20.00 (operando 'b') e 5.00 (operando 'a').

- Calcula: 5.00 * 20.00 = 100.00

- Empilha o resultado 100.00.

Pilha atual: [9.00 100.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 100.00 (operando 'b') e 9.00 (operando 'a').

- Calcula: 9.00 + 100.00 = 109.00

- Empilha o resultado 109.00.

Pilha atual: [109.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 109

Obs: Foi observado que na forma infixa posta na tabela contida no anexo do documebto, em menão ao Teste 05, que o processo não segue ordem de conversão correta da expressão posfixa 9 5 2 8 * 4 + * + para infixa 9 + (5 * (2 + 8 * 4)). A primeira tem resultado 109, e a segunda 179. Enquanto a forma correta convertida 9 + (5 * ((2 * 8) + 4)) tem resultado condizente 109.

==========

Entrada - ANÁLISE DA EXPRESSAO: 9 + (5 * ((2 * 8) + 4))

==========

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---

Expressão identificada como: INFIXA.

Convertendo para a forma Pos-fixa para o cálculo.

- Original (Infixa): 9 + (5 * ((2 * 8) + 4))

- Convertida (Pos-fixa): 9 5 2 8 * 4 + * +

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "9"

-> É um número. Empilha 9.

Pilha atual: [9.00]

Lendo token: "5"

-> É um número. Empilha 5.

Pilha atual: [9.00 5.00]

Lendo token: "2"

-> É um número. Empilha 2.

Pilha atual: [9.00 5.00 2.00]

Lendo token: "8"

-> É um número. Empilha 8.

Pilha atual: [9.00 5.00 2.00 8.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 8.00 (operando 'b') e 2.00 (operando 'a').

- Calcula: 2.00 * 8.00 = 16.00

- Empilha o resultado 16.00.

Pilha atual: [9.00 5.00 16.00]

Lendo token: "4"

-> É um número. Empilha 4.

Pilha atual: [9.00 5.00 16.00 4.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 16.00 (operando 'a').

- Calcula: 16.00 + 4.00 = 20.00

- Empilha o resultado 20.00.

Pilha atual: [9.00 5.00 20.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 20.00 (operando 'b') e 5.00 (operando 'a').

- Calcula: 5.00 * 20.00 = 100.00

- Empilha o resultado 100.00.

Pilha atual: [9.00 100.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 100.00 (operando 'b') e 9.00 (operando 'a').

- Calcula: 9.00 + 100.00 = 109.00

- Empilha o resultado 109.00.

Pilha atual: [109.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 109

```
============================================================
    Entrada -  ANÁLISE DA EXPRESSAO: 9 + (5 *(2 + 8 * 4))
============================================================

--- ETAPA 1: CONVERSAO DE NOTACAO ---
Expressão identificada como: INFIXA.
Convertendo para a forma Pos-fixa para o calculo.
 - Original (Infixa):   9 + (5 *(2 + 8 * 4))
 - Convertida (Pos-fixa): 9 5 2 8 4 * + * +

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---
O cálculo e feito lendo cada item (token) da expressão pos-fixa.
números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "9"
 -> É um número. Empilha 9.
    Pilha atual: [ 9.00 ]

Lendo token: "5"
 -> É um número. Empilha 5.
    Pilha atual: [ 9.00 5.00 ]

Lendo token: "2"
 -> É um número. Empilha 2.
    Pilha atual: [ 9.00 5.00 2.00 ]

Lendo token: "8"
 -> É um número. Empilha 8.
    Pilha atual: [ 9.00 5.00 2.00 8.00 ]

Lendo token: "4"
 -> É um número. Empilha 4.
    Pilha atual: [ 9.00 5.00 2.00 8.00 4.00 ]
```

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 4.00 (operando 'b') e 8.00 (operando 'a').

- Calcula: 8.00 * 4.00 = 32.00

- Empilha o resultado 32.00.

Pilha atual: [9.00 5.00 2.00 32.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 32.00 (operando 'b') e 2.00 (operando 'a').

- Calcula: 2.00 + 32.00 = 34.00

- Empilha o resultado 34.00.

Pilha atual: [9.00 5.00 34.00]

Lendo token: "*"

-> É um operador. Requer 2 operandos.

- Desempilha 34.00 (operando 'b') e 5.00 (operando 'a').

- Calcula: 5.00 * 34.00 = 170.00

- Empilha o resultado 170.00.

Pilha atual: [9.00 170.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 170.00 (operando 'b') e 9.00 (operando 'a').

- Calcula: 9.00 + 170.00 = 179.00

- Empilha o resultado 179.00.

Pilha atual: [179.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 179

3.5 Teste n° 05

==========

Entrada - ANÁLISE DA EXPRESSAO: 2 3 + log 5 /

==========

Saída -

--- ETAPA 1: CONVERSAO DE NOTACAO ---

Expressão identificada como: POS-FIXA.

Convertendo para a forma Infixa para visualização.

- Original (Pos-fixa): 2 3 + log 5 /

- Convertida (Infixa): log((2 + 3)) / 5

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

Números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "2"

-> É um número. Empilha 2.

Pilha atual: [2.00]

Lendo token: "3"

-> É um número. Empilha 3.

Pilha atual: [2.00 3.00]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 3.00 (operando 'b') e 2.00 (operando 'a').

- Calcula: 2.00 + 3.00 = 5.00

- Empilha o resultado 5.00.

Pilha atual: [5.00]

Lendo token: "log"

-> É uma função. Requer 1 operando.

- Desempilha 5.00 (operando 'a').

- Calcula: log(5.00) = 0.698970

- Empilha o resultado 0.698970.

Pilha atual: [0.70]

Lendo token: "5"

-> É um número. Empilha 5.

Pilha atual: [0.70 5.00]

Lendo token: "/"

-> É um operador. Requer 2 operandos.

- Desempilha 5.00 (operando 'b') e 0.70 (operando 'a').

- Calcula: 0.70 / 5.00 = 0.14

- Empilha o resultado 0.14.

Pilha atual: [0.14]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 0.139794

3.6 Teste n° 06

==========

Entrada - ANÁLISE DA EXPRESSAO: 0.5 45 sen 2 ^ +

==========

Saída -

--- ETAPA 1: CONVERSÃO DE NOTACAO ---

Expressão identificada como: POS-FIXA.

Convertendo para a forma Infixa para visualização.

- Original (Pos-fixa): 0.5 45 sen 2 ^ +

- Convertida (Infixa): 0.5 + (sen(45) ^ 2)

--- ETAPA 2: CÁLCULO DO RESULTADO (a partir da forma Pos-fixa) ---

O cálculo e feito lendo cada item (token) da expressão pos-fixa.

números são empilhados. Operadores desempilham valores, calculam e empilham o resultado.

Lendo token: "0.5"

-> É um número. Empilha 0.5.

Pilha atual: [0.50]

Lendo token: "45"

-> É um número. Empilha 45.

Pilha atual: [0.50 45.00]

Lendo token: "sen"

-> É uma função. Requer 1 operando.

- Desempilha 45.00 (operando 'a').

- Calcula: sen(45.00) = 0.707107

- Empilha o resultado 0.707107.

Pilha atual: [0.50 0.71]

Lendo token: "2"

-> É um número. Empilha 2.

Pilha atual: [0.50 0.71 2.00]

Lendo token: "^"

-> É um operador. Requer 2 operandos.

- Desempilha 2.00 (operando 'b') e 0.71 (operando 'a').

- Calcula: 0.71 ^ 2.00 = 0.50

- Empilha o resultado 0.50.

Pilha atual: [0.50 0.50]

Lendo token: "+"

-> É um operador. Requer 2 operandos.

- Desempilha 0.50 (operando 'b') e 0.50 (operando 'a').

- Calcula: 0.50 + 0.50 = 1.00

- Empilha o resultado 1.00.

Pilha atual: [1.00]

Fim da expressão. O resultado final e o último valor na pilha.

>>> RESULTADO FINAL: 1

Obs: Aqui ocorre o mesmo erro do TESTE 4. A conversão da Pos-fixa 0.5 45 sen 2 ^ + é 0.5 + (sen(45) ^ 2), e não condizente com a conversão Infixa na tabela sen(45) ^2 + 0,5. Todas possuem resultado 1, mas sua notação não é condizente.

4. Conclusão

O desenvolvimento deste trabalho permitiu a aplicação prática e o aprofundamento dos conhecimentos sobre estruturas de dados, em especial o TAD Pilha, na resolução de um problema clássico da ciência da computação. O resultado foi um programa funcional e robusto, capaz de interpretar, converter e avaliar expressões matemáticas complexas de forma correta e consistente.

A principal dificuldade encontrada durante a implementação foi a correta manipulação da precedência e associatividade de operadores no algoritmo Shunting-Yard, além do tratamento de casos especiais como operadores unários (ex: -5) em contraste com a subtração binária. A análise da tabela de testes proposta no enunciado também revelou certas inconsistências entre as notações infixa e pós-fixa, o que exigiu uma decisão de projeto para priorizar a correção matemática do algoritmo em detrimento replicação exata de uma conversão inconsistente.

Como possíveis melhorias para uma versão futura, podem ser implementadas:

- **Validação Sintática Abrangente:** Implementar uma verificação mais rigorosa da estrutura da expressão antes do cálculo. Isso permitiria identificar e reportar erros específicos, como parênteses que não fecham, operadores em posições inválidas (ex: 5 * + 3) ou falta de operandos.

- **Reconhecimento de Tokens Inválidos:** Aprimorar a fase de tokenização para que o programa recuse ativamente caracteres ou sequências inválidas (ex: a_b, $$, 3.4.5).

- **Suporte a Variáveis:** Permitir que o usuário defina e utilize variáveis em suas expressões (ex: a = 5, b = 10, a + b), transformando a calculadora em um interpretador matemático simples.

- **Flexibilidade de Operações**: Melhorar o tratamento de números para aceitar tanto o ponto (.) quanto a vírgula (,) como separadores com suas características diferentes dentro da Matemática de forma inteligente, aumentando a usabilidade do programa.

- **Modificação para acentuação em diferentes línguas existentes**.

Referências

CÓDIGO FONTE TV. Estrutura de Dados (A famosa ED que todo dev tem que aprender) // Dicionário do Programador. [S. l.]: [s. n.], 2020. 1 vídeo (12 min 4 s). Disponível em: *https://www.youtube.com/watch?v=EfF1M7myAyY*. Acesso em: 20 jun. 2025.

CORMEN, Thomas H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. Algoritmos: teoria e prática. 3. ed. Rio de Janeiro: Elsevier, 2012.

DE ALUNO PARA ALUNO. Pilha | Avaliando Expressões Pós-Fixas | Estrutura de Dados. [S. l.]: [s. n.], 2020. 1 vídeo (24 min 30 s). Disponível em: *https://www.youtube.com/watch?v=vGTotPSx0ZY*. Acesso em: 20 jun. 2025.

EUSTÁQUIO, Marcelo. Avaliador de expressões numéricas. Enunciado do Trabalho Prático, Disciplina de Estrutura de Dados. Brasília: Universidade Católica de Brasília, 2025.

GEEKSFORGEEKS. Shunting Yard Algorithm. Disponível em: *https://www.geeksforgeeks.org/*. Acesso em: 20 jun. 2025.

KNUTH, Donald. The Art of Computer Programming. 3. ed. Boston: Addison-Wesley, 1997.

PROJETO PANDA. Expressões Infixa, Prefixa e Pós-fixa. São Paulo: IME-USP. Disponível em: *https://panda.ime.usp.br/panda/static/pythonds_pt/03-EDBasicos/09-ExpressoesInfixaPrefixaPosfixa.html*. Acesso em: 20 jun. 2025.

PROFESSOR ISIDRO. Explicando Estrutura de Dados - Pilha. [S. l.]: [s. n.], 2016. 1 vídeo (7 min 24 s). Disponível em: *https://www.youtube.com/watch?v=2V91Re1czwA*. Acesso em: 20 jun. 2025.

REIS, Eliezer Souza. Algoritmo Shunting Yard. [S. l.]: [s. n.], 2018. 1 vídeo (23 min 1 s). Disponível em: *https://www.youtube.com/watch?v=RRCwjwlLViU*. Acesso em: 20 jun. 2025.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. Estruturas de Dados Usando C. São Paulo: Pearson Makron Books, 1995.

WIKIPÉDIA. Notação pós-fixa. Disponível em: *https://pt.wikipedia.org/wiki/Nota%C3%A7%C3%A3o_polonesa_inversa*. Acesso em: 20 jun. 2025.

Anexos

TP03 - Avaliador de expressões numéricas.pdf

Estrutura de Dados usando C.pdf

Algoritmos: teoria e prática

calculadora.h

(Professor, destacamos unicamente essa parte em vermelho devido a confusão de nomenclatura. No documento o Sr. Pede calculadora.H, mas no seu modelo, está define EXPRESSAO.H. Utilizamos o nome no documento(calculadora.h).

```
#ifndef EXPRESSAO_H // CALCULADORA_H
#define EXPRESSAO_H //CALCULADORA_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 <string.h>
#include <ctype.h>
#include <math.h>
#include "calculadora.h"

#define MAX 100
#ifndef M_PI
#define M_PI 3.14159265358979323846
#endif

// ------------------------- Estruturas de Pilha ------------------------- //

typedef struct {
   char items[MAX][256];
   int top;
} StackStr;

typedef struct {
   float items[MAX];
   int top;
} StackFloat;

// ------------------------- Funções de Pilha ------------------------- //

void initStr(StackStr* s) { s->top = -1; }
int emptyStr(StackStr* s) { return s->top == -1; }
```

```
void pushStr(StackStr* s, char* val) {
   if (s->top < MAX - 1) {
      strcpy(s->items[++s->top], val);
   }
}
char* popStr(StackStr* s) {
   if (!emptyStr(s)) {
      return s->items[s->top--];
   }
   return "";
}
char* peekStr(StackStr* s) { return s->items[s->top]; }

void initFloat(StackFloat* s) { s->top = -1; }
void pushFloat(StackFloat* s, float val) { s->items[++s->top] = val; }
float popFloat(StackFloat* s) { return s->items[s->top--]; }
int sizeFloat(StackFloat* s) { return s->top + 1; }

// ------------------------- Funções Auxiliares ------------------------- //

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

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

int precedence(char* op) {
   if (strcmp(op, "^") == 0) return 4;
   if (isFunction(op)) return 4;
```

```
    if (strcmp(op, "*") == 0 || strcmp(op, "/") == 0 || strcmp(op, "%") == 0) return 3;
    if (strcmp(op, "+") == 0 || strcmp(op, "-") == 0) return 2;
    return 0;
}

int isLeftAssociative(char* op) {
    return strcmp(op, "^") != 0;
}

int isNumeric(char* s) {
    if (!s || s[0] == '\0') return 0;
    char *end;
    strtod(s, &end);
    return *end == '\0';
}

int tokenize(char* expr, char tokens[][20]) {
    int i = 0, j = 0, k = 0;
    while (expr[i]) {
        if (isspace(expr[i])) { i++; continue; }
        if (isdigit(expr[i]) || (expr[i] == '.' && isdigit(expr[i+1])) || ((expr[i] == '-' || expr[i] == '+') &&
isdigit(expr[i+1]) && (i == 0 || strchr("([{^*/+-%", expr[i-1])))) {
            j = 0;
            if (expr[i] == '-' || expr[i] == '+') tokens[k][j++] = expr[i++];
            while (isdigit(expr[i]) || expr[i] == '.') tokens[k][j++] = expr[i++];
            tokens[k++][j] = '\0';
        } else if (isalpha(expr[i])) {
            j = 0;
            while (isalpha(expr[i])) tokens[k][j++] = expr[i++];
            tokens[k++][j] = '\0';
        } else {
            tokens[k][0] = expr[i++];
            tokens[k++][1] = '\0';
```

```
        }
    }
    return k;
}

// --- FUNÇÃO DE VALIDAÇÃO (Restaurada para detecção robusta) ---

int validatePostfix(char tokens[][20], int n) {
    StackFloat stack;
    initFloat(&stack);

    if (n == 0) return 0;

    for (int i = 0; i < n; i++) {
        if (isNumeric(tokens[i])) {
            pushFloat(&stack, 1.0); // Empilha um valor qualquer
        } else if (isOperator(tokens[i])) {
            if (sizeFloat(&stack) < 2) return 0; // Inválido
            popFloat(&stack);
            popFloat(&stack);
            pushFloat(&stack, 1.0);
        } else if (isFunction(tokens[i])) {
            if (sizeFloat(&stack) < 1) return 0; // Inválido
            popFloat(&stack);
            pushFloat(&stack, 1.0);
        } else if (strcmp(tokens[i], "(") == 0 || strcmp(tokens[i], ")") == 0) {
            return 0; // Expressões pós-fixas não têm parênteses
        } else {
            return 0; // Token inválido
        }
    }

    // Se no final sobrar exatamente um elemento na pilha, a expressão é válida
```

```
    return (sizeFloat(&stack) == 1);
}

// --- FUNÇÃO DE DETECÇÃO (Para ser chamada pelo main.c) ---

// Retorna 1 se for Pós-fixa, 0 se for Infixa

int detectarNotacao(char *expr) {
    char tokens[100][20];
    int n = tokenize(expr, tokens);

    if (validatePostfix(tokens, n)) {
        return 1;
    }
    return 0;
}



// --- FUNÇÕES DE CONVERSÃO E CÁLCULO ---

char *getFormaInFixa(char *Str) {
    static char resultado[512];
    char tokens[100][20];
    int n = tokenize(Str, tokens);
    StackStr s;
    initStr(&s);
    for (int i = 0; i < n; i++) {
        if (isNumeric(tokens[i])) {
            pushStr(&s, tokens[i]);
        } else if (isOperator(tokens[i])) {
            if (s.top < 1) return "ERRO: Faltam operandos";
            char b[128], a[128];
            strcpy(b, popStr(&s));
            strcpy(a, popStr(&s));
```

```
            char temp[256];
            sprintf(temp, "(%s %s %s)", a, tokens[i], b);
            pushStr(&s, temp);
        } else if (isFunction(tokens[i])) {
            if (s.top < 0) return "ERRO: Faltam operandos para funcao";
            char a[128];
            strcpy(a, popStr(&s));
            char temp[256];
            sprintf(temp, "%s(%s)", tokens[i], a);
            pushStr(&s, temp);
        }
    }
    if (s.top != 0) return "ERRO: Expressao malformada";
    strcpy(resultado, popStr(&s));
    if (resultado[0] == '(' && resultado[strlen(resultado) - 1] == ')') {
        resultado[strlen(resultado) - 1] = '\0';
        strcpy(resultado, resultado + 1);
    }
    return resultado;
}

char *getFormaPosFixa(char *Str) {
    static char resultado[512];
    char tokens[100][20], output[100][20];
    int n = tokenize(Str, tokens), outSize = 0;
    StackStr opStack;
    initStr(&opStack);
    for (int i = 0; i < n; i++) {
        if (isNumeric(tokens[i])) strcpy(output[outSize++], tokens[i]);
        else if (isFunction(tokens[i])) pushStr(&opStack, tokens[i]);
        else if (isOperator(tokens[i])) {
            while (!emptyStr(&opStack) && strcmp(peekStr(&opStack), "(") != 0 &&
                (precedence(peekStr(&opStack)) > precedence(tokens[i]) ||
```

```
            (precedence(peekStr(&opStack)) == precedence(tokens[i]) && isLeftAssociative(tokens[i]))) {
            strcpy(output[outSize++], popStr(&opStack));
        }
        pushStr(&opStack, tokens[i]);
    } else if (strcmp(tokens[i], "(") == 0) pushStr(&opStack, tokens[i]);
    else if (strcmp(tokens[i], ")") == 0) {
        while (!emptyStr(&opStack) && strcmp(peekStr(&opStack), "(") != 0) strcpy(output[outSize++],
popStr(&opStack));
        if (!emptyStr(&opStack)) popStr(&opStack);
        if (!emptyStr(&opStack) && isFunction(peekStr(&opStack))) strcpy(output[outSize++],
popStr(&opStack));
    }
  }
  while (!emptyStr(&opStack)) strcpy(output[outSize++], popStr(&opStack));
  resultado[0] = '\0';
  for (int i = 0; i < outSize; i++) {
    strcat(resultado, output[i]);
    if (i < outSize - 1) strcat(resultado, " ");
  }
  return resultado;
}

float getValorPosFixa(char *StrPosFixa) {
  char tokens[100][20];
  int n = tokenize(StrPosFixa, tokens);
  StackFloat s;
  initFloat(&s);
  for (int i = 0; i < n; i++) {
    if (isNumeric(tokens[i])) pushFloat(&s, atof(tokens[i]));
    else if (isOperator(tokens[i])) {
      if (sizeFloat(&s) < 2) return NAN;
      float b = popFloat(&s), a = popFloat(&s), r = 0.0;
      if (strcmp(tokens[i], "+") == 0) r = a + b;
      else if (strcmp(tokens[i], "-") == 0) r = a - b;
```

```
            else if (strcmp(tokens[i], "*") == 0) r = a * b;
            else if (strcmp(tokens[i], "/") == 0) { if (b == 0) return NAN; r = a / b; }
            else if (strcmp(tokens[i], "%") == 0) { if (b == 0) return NAN; r = fmod(a, b); }
            else if (strcmp(tokens[i], "^") == 0) r = pow(a, b);
            pushFloat(&s, r);
        } else if (isFunction(tokens[i])) {
            if (sizeFloat(&s) < 1) return NAN;
            float a = popFloat(&s), r = 0.0;
            if (strcmp(tokens[i], "sen") == 0) r = sin(a * M_PI / 180.0);
            else if (strcmp(tokens[i], "cos") == 0) r = cos(a * M_PI / 180.0);
            else if (strcmp(tokens[i], "tg") == 0) r = tan(a * M_PI / 180.0);
            else if (strcmp(tokens[i], "log") == 0) { if (a <= 0) return NAN; r = log10(a); }
            else if (strcmp(tokens[i], "raiz") == 0) { if (a < 0) return NAN; r = sqrt(a); }
            pushFloat(&s, r);
        }
    }
    if (sizeFloat(&s) != 1) return NAN;
    return popFloat(&s);
}

float getValorInFixa(char *StrInFixa) {
    char *posfixa = getFormaPosFixa(StrInFixa);
    if(strstr(posfixa, "ERRO") != NULL) return NAN;
    return getValorPosFixa(posfixa);
}
```

main.c

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

// Protótipos das funções que estão em calculadora.c

int isOperator(char* token);
int isFunction(char* token);
int isNumeric(char* s);
int tokenize(char* expr, char tokens[][20]);
int detectarNotacao(char *expr);

// ADICIONADO: Protótipo para a função de explicação

void calcularComExplicacao(char* posFixaStr);



// Função para imprimir o estado atual da pilha de floats

void imprimirPilha(float* stack, int top) {
   printf("     Pilha atual: [ ");
   for (int i = 0; i <= top; i++) {
      printf("%.2f ", stack[i]);
   }
   printf("]\n");
```

```
}

// Função que gerencia todo o processo para uma expressão

void processarEExplicar(char* expressao) {

   // Pré-processamento: remove vírgulas e converte para minúsculas

   for (int i = 0; expressao[i]; i++) {
      if (expressao[i] == ',') expressao[i] = '.';
      expressao[i] = tolower(expressao[i]);
   }

   // Chama a função robusta de detecção

   int isPosfixa = detectarNotacao(expressao);

   printf("\n================================================================\n");
   printf("            ANALISE DA EXPRESSAO: %s\n", expressao);
   printf("================================================================\n");

   char *convertida;
   char *paraCalcular;

   printf("\n--- ETAPA 1: CONVERSAO DE NOTACAO ---\n");
   if (isPosfixa) {
      printf("Expressao identificada como: POS-FIXA.\n");
      printf("Convertendo para a forma Infixa para visualizacao.\n");
      printf("  - Original (Pos-fixa): %s\n", expressao);
      convertida = getFormaInFixa(expressao);
      printf("  - Convertida (Infixa): %s\n", convertida);
      paraCalcular = expressao;
   } else {
      printf("Expressao identificada como: INFIXA.\n");
```

```
        printf("Convertendo para a forma Pos-fixa para o calculo.\n");
        printf("  - Original (Infixa):   %s\n", expressao);
        convertida = getFormaPosFixa(expressao);
        printf("  - Convertida (Pos-fixa): %s\n", convertida);
        paraCalcular = convertida;
    }

    if(strstr(convertida, "ERRO") != NULL) {
        printf("\nERRO na conversao. Verifique a expressao.\n");
        return;
    }

    calcularComExplicacao(paraCalcular);
}


// --- FUNÇÃO DE CÁLCULO COM EXPLICAÇÃO ---


void calcularComExplicacao(char* posFixaStr) {
    printf("\n--- ETAPA 2: CALCULO DO RESULTADO (a partir da forma Pos-fixa) ---\n");
    printf("O calculo e feito lendo cada item (token) da expressao pos-fixa.\nNumeros sao empilhados.
Operadores desempilham valores, calculam e empilham o resultado.\n");

    char tokens[100][20];
    int n = tokenize(posFixaStr, tokens);

    float stackItems[100];
    int top = -1;

    for (int i = 0; i < n; i++) {
        printf("\nLendo token: \"%s\"\n", tokens[i]);
        if (isNumeric(tokens[i])) {
            float val = atof(tokens[i]);
```

```
        stackItems[++top] = val;
        printf("  -> E um numero. Empilha %g.\n", val);
    } else if (isOperator(tokens[i])) {
        printf("  -> E um operador. Requer 2 operandos.\n");
        if (top < 1) { printf("ERRO: Faltam operandos na pilha!\n"); return; }
        float b = stackItems[top--];
        float a = stackItems[top--];
        float r = 0.0;
        printf("     - Desempilha %.2f (operando 'b') e %.2f (operando 'a').\n", b, a);

        if (strcmp(tokens[i], "+") == 0) { r = a + b; printf("     - Calcula: %.2f + %.2f = %.2f\n", a, b, r); }
        else if (strcmp(tokens[i], "-") == 0) { r = a - b; printf("     - Calcula: %.2f - %.2f = %.2f\n", a, b, r); }
        else if (strcmp(tokens[i], "*") == 0) { r = a * b; printf("     - Calcula: %.2f * %.2f = %.2f\n", a, b, r); }
        else if (strcmp(tokens[i], "/") == 0) { r = a / b; printf("     - Calcula: %.2f / %.2f = %.2f\n", a, b, r); }
        else if (strcmp(tokens[i], "%") == 0) { r = fmod(a,b); printf("     - Calcula: %.2f %% %.2f = %.2f\n", a,
b, r); }
        else if (strcmp(tokens[i], "^") == 0) { r = pow(a,b); printf("     - Calcula: %.2f ^ %.2f = %.2f\n", a, b,
r); }

        stackItems[++top] = r;
        printf("     - Empilha o resultado %.2f.\n", r);
    } else if (isFunction(tokens[i])) {
        printf("  -> E uma funcao. Requer 1 operando.\n");
        if (top < 0) { printf("ERRO: Falta operando na pilha!\n"); return; }
        float a = stackItems[top--];
        float r = 0.0;
        printf("     - Desempilha %.2f (operando 'a').\n", a);

        if (strcmp(tokens[i], "sen") == 0) { r = sin(a * M_PI/180.0); printf("     - Calcula: sen(%.2f) = %f\n", a,
r); }
        else if (strcmp(tokens[i], "cos") == 0) { r = cos(a * M_PI/180.0); printf("     - Calcula: cos(%.2f) =
%f\n", a, r); }
        else if (strcmp(tokens[i], "tg") == 0) { r = tan(a * M_PI/180.0); printf("     - Calcula: tg(%.2f) = %f\n",
a, r); }
```

```
            else if (strcmp(tokens[i], "log") == 0) { r = log10(a); printf("     - Calcula: log(%.2f) = %f\n", a, r); }
            else if (strcmp(tokens[i], "raiz") == 0) { r = sqrt(a); printf("     - Calcula: raiz(%.2f) = %f\n", a, r); }

            stackItems[++top] = r;
            printf("     - Empilha o resultado %f.\n", r);
        }
        imprimirPilha(stackItems, top);
    }
    printf("\n----------------------------------------------------------------\n");
    if(top == 0) {
        printf("Fim da expressao. O resultado final e o ultimo valor na pilha.\n");
        printf(">>> RESULTADO FINAL: %f\n", stackItems[top]);
    } else {
        printf("ERRO: A expressao e invalida. Sobraram %d elementos na pilha.\n", top + 1);
    }
}

void mostrarMenu() {
    printf("\n=== AVALIADOR DE EXPRESSOES MATEMATICAS ===\n");
    printf("1. Converter e calcular expressao\n");
    printf("2. Sair\n");
    printf("Escolha uma opcao: ");
}

int main() {
    char expressao[512];
    int opcao;

    do {
        mostrarMenu();
        if (scanf("%d", &opcao) != 1) {
            while(getchar() != '\n');
            opcao = 0;
```

```
        }
        while (getchar() != '\n');

        switch (opcao) {
            case 1: {
                printf("\nDigite a expressao (ex: (3+4)*5 ou 3 4 + 5 *): ");
                fgets(expressao, sizeof(expressao), stdin);
                expressao[strcspn(expressao, "\n")] = 0;

                if (strlen(expressao) == 0) {
                    printf("Expressao vazia!\n");
                    break;
                }
                processarEExplicar(expressao);
                break;
            }
            case 2:
                printf("\nSaindo do programa...\n");
                break;
            default:
                printf("\nOpcao invalida! Tente novamente.\n");
        }
    } while (opcao != 2);

    return 0;
}
```