|  |  |
| --- | --- |
|  | **Universidade do Estado da Bahia**  **Estrutura de Dados I**  **Prof. Carlos Helano** |

**Objetivo**

Desenvolver o programa na linguagem C conforme especificação a seguir, usando as bibliotecas e os conceitos trabalhados em Estrutura de Dados.

**Organização**

O trabalho pode ser desenvolvido em equipes de 2 ou 3 componentes. O programa desenvolvido deve ser postado no Teams e apresentado pela equipe durante o horário da aula na data marcada.

**Cenário**

O Trabalho consiste em desenvolver um programa capaz de realizar cálculos de expressões numéricas utilizando os conceitos de Estrutura de Dados trabalhados no semestre. Os próximos parágrafos irão explicar melhor o contexto e orientar melhor como o trabalho deve ser desenvolvido.

Estamos tão acostumados a digitar expressões e receber resultados, que raramente paramos para pensar sobre o que deve acontecer dentro do computador, a partir do momento em que fornecemos a expressão, até o momento em que recebemos a resposta. Para que tenha sucesso na implementação da solução para este trabalho passaremos algumas orientações e estratégias que devem ser seguidas com a aplicação da estrutura de dados em pilha.

**Forma das Expressões Numéricas**

Para simplificar o entendimento, vamos nos restringir a expressões aritméticas compostas exclusivamente pelos seguintes elementos:

* Identificadores: 0..9
* Operadores: +, -, \*, /
* Parênteses: ( )

Exemplos de algumas expressões:

* 2+3
* 3-2\*4
* 8/(2\*(4-1)+5)

Os identificadores representam os valores numéricos numa expressão.

As operações devem ser efetuadas de acordo com a ordem determinada pelas regras usuais da aritmética: multiplicação e divisão têm prioridade (precedência) sobre adição e subtração, sendo que os operadores com a mesma precedência são efetuados na ordem em que aparecem na expressão, da esquerda para a direita.

Os parênteses são empregados para alterar a ordem natural de avaliação dos operadores numa certa expressão. Por exemplo, na expressão 2\*(4+1), os parênteses forçam a adição ser efetuada antes da multiplicação, mesmo tendo esta última uma prioridade maior.

Assim uma expressão hipotética:

A-B+C/D\*E deve ser entendida como ((A-B)+((C/D)\*E)), mas neste caso os parênteses seriam desnecessários.

Portanto, dois fatores contribuem para dificultar a avaliação de uma expressão:

* A existência de prioridades diferentes para os operadores, que não nos permite efetuá-los na ordem em que os encontramos na expressão;
* A existência de parênteses, que alteram a prioridade das operações.

**Notação Polonesa Reversa**

Estamos habituados a escrever expressões numéricas sempre colocando o operador entre os valores que ele deve operar. Ao representar a soma de dois valores A e B, normalmente escrevemos A+B. Entretanto, esta não é a única forma possível. Existem na verdade 3 formas:

* Infixa : A + B
* Prefixa : + A B (também chamada de **Notação Polonesa**)
* Posfixa : A B + (também chamada de **Notação Polonesa Reversa**)

A vantagem em se usar a notação polonesa reversa é que, como cada operador aparece imediatamente após os valores que ele deve operar, a ordem em que ele aparece é também a ordem em que deve ser efetuado. E dessa forma, não existe a necessidade de se usar nenhum parêntese.

|  |  |
| --- | --- |
| **Notação Infixa** | **Notação Posfixa** |
| A + B \* C | A B C \* + |
| A \* (B + C) | A B C + \* |
| (A + B) / (C – D) | A B + C D - / |
| (A + B) / (C – D) \* E | A B + C D - / E \* |

Para resolver o problema vamos dividir o trabalho em etapas. Diante do que foi descrito, já foi possível perceber que uma expressão que esteja totalmente preenchida com parênteses é relativamente mais fácil de desenvolver uma lógica.

Outro ponto importante é que o uso da notação polonesa facilita o desenvolvimento de uma lógica para realizar os cálculos.

Por isso, na primeira etapa, vamos nos preocupar apenas em gerar a expressão em notação polonesa, a partir do input do usuário de uma expressão normal (infixa).

**Etapa 1 - Gerar a Expressão Posfixa – Versão Simplificada**

Nesta etapa, vamos assumir que o input do usuário esteja com todos os parênteses preenchidos. Até mesmo com os parênteses que teoricamente são desnecessários.

Vamos utilizar com exemplo as expressões da tabela anterior.

|  |  |  |
| --- | --- | --- |
| **Notação Infixa** | **Notação Infixa (Parênteses)** | **Notação Posfixa** |
| A + B \* C | (A + (B \* C)) | A B C \* + |
| A \* (B + C) | (A \* (B + C)) | A B C + \* |
| (A + B) / (C – D) | ((A + B) / (C – D)) | A B + C D - / |
| (A + B) / (C – D) \* E | ((A + B) / (C – D) \* E) | A B + C D - / E \* |

O trecho de código a seguir com a função posfixa\_simples deve gerar a expressão Posfixa da forma desejada.

Para verificar o resultado basta exibir a pilha plOutput na função main() do programa.

enum Operacoes {

MULT='\*',

SOMA='+',

SUB='-',

DIV='/'

};

void posfixa\_simples(T\_Pilha \*plOutput, char exp[])

{

T\_Pilha plOperadores;

iniciarPilha(&plOperadores);

T\_Item item;

for (int i = 0; i < strlen(exp); i++)

{

item.campo = exp[i];

switch (exp[i])

{

case '0'...'9' : inserir(plOutput, item); break;

case MULT...DIV : inserir(&plOperadores, item); break;

case ')' :

{

inserir(plOutput, plOperadores.dados[plOperadores.topo-1]);

remover(&plOperadores);

break;

}

}

}

}

**Etapa 2 - Calcular a Expressão – Expressão simplificada**

Nesta etapa devemos realizar o cálculo da expressão utilizando a expressão gerada na etapa anterior. Exemplo :

Expressão infixa inserida pelo usuário : **(1+(2\*3))**

Expressão posfixa (plOutput) gerada : **1 2 3 \* +**

Devemos percorrer toda a pilha plOutput da base até o topo e testar cada caracter.

* Se item (caracter) for igual a um dos identificadores [0..9] inserir na pilha plResult
* Se item (caracter) for igual a um operador [+,-,\*,/]
  + Remover do topo os dois últimos valores da pilha plResult
  + Inserir na pilha plResult o valor resultante da operação entre os dois últimos valores.

|  |  |  |  |
| --- | --- | --- | --- |
| **i** | **Item** | **Ação** | **Pilha Resultado (plResult)** |
| 0 | 1 | Inserir valor 1 na plResult | [1] |
| 1 | 2 | Inserir valor 2 na plResult | [1,2] |
| 2 | 3 | Inserir valor 2 na plResult | [1,2,3] |
| 3 | \* | Remover o valor 3 da plResult  Remover o valor 2 da plResult  Inserir o valor 6 (resultado da operação) na plResult | [1,6] |
| 4 | + | Remover o valor 6 da plResult  Remover o valor 1 da plResult  Inserir o valor 7 (resultado da operação) na plResult | [7] |

**Etapa 3 – Evolução da Solução**

Nesta etapa será necessário evoluir a solução desenvolvida até o momento, pois algumas questões não foram contempladas

* Mesmo considerando a expressão simplificada (com parênteses) a solução implementada ainda tem problemas.
  + A expressão só funciona com identificadores numéricos de 0..9
  + Caso o resultado de alguma operação seja de mais de um dígito, a solução também não deve funcionar. Exemplo : Na expressão infixa (2+(3\*4)) teremos a expressão posfixa 2 3 4 \* + , e o resultado da primeira operação já causaria problema (3\*4 = 12).
* A versão evoluída do programa deve contemplar o input de uma expressão sem a obrigatoriedade dos parênteses e adotar a prioridade das operações conforme o regramento
* A solução também não contempla valores reais na expressão (Ex: 2.1 ; 3,14 )
* A solução também não contempla dígitos numéricos negativos.

**Avaliação**

A solução implementada de forma correta na Etapa 1 garante até 5 pontos na avaliação deste trabalho.

A solução implementada de forma correta nas Etapas 1 e 2 garante até 7 pontos na avaliação deste trabalho.

A solução implementada de forma correta na Etapa 3 para os 2 primeiros tópicos garante a nota 10

A solução implementada de forma correta na Etapa 3 para os 2 últimos tópicos garante um ponto extra