



**UNIVERSIDADE DO VALE DO ITAJAÍ**

ESCOLA POLITÉCNICA  
ESTRUTURA DE DADOS

FILIPPI SCALVIN DA COSTA  
JOÃO PEDRO PRUDÊNCIO

**PROJETO M1:**  
RESOLUÇÃO DE EXPRESSÃO CONVERTIDA PARA NOTAÇÃO POLONESA  
INVERSA

ITAJAÍ – SC  
2023

FILIPPI SCALVIN DA COSTA  
JOÃO PEDRO PRUDÊNCIO

**PROJETO M1:**  
RESOLUÇÃO DE EXPRESSÃO CONVERTIDA PARA NOTAÇÃO POLONESA  
INVERSA

Trabalho acadêmico apresentado na Universidade do Vale do Itajaí, como requisito parcial à obtenção de nota no componente curricular M1 apresentado na disciplina de Estrutura de Dados, referente ao curso de Ciência da Computação.

Professor(a): Marcos Carrard.

ITAJAÍ - SC  
2023

## SUMÁRIO

<b>SUMÁRIO .....</b>	<b>IV</b>
<b>1. OBJETIVOS.....</b>	<b>5</b>
<b>3. RESUMO.....</b>	<b>6</b>
<b>4. DESENVOLVIMENTO .....</b>	<b>8</b>
<b>4.1 IMPLEMENTAÇÃO DO ALGORITMO.....</b>	<b>9</b>
<b>4.1.1 FUNÇÃO AUXILIAR – É OPERAÇÃO? .....</b>	<b>11</b>
<b>4.1.2 FUNÇÃO AUXILIAR - MENOR PRECEDENTE .....</b>	<b>12</b>
<b>4.1.3 FUNÇÃO AUXILIAR – É OPERANDO? .....</b>	<b>13</b>
<b>4.1.4 FUNÇÃO AUXILIAR – CALCULAR EXPRESSÃO .....</b>	<b>14</b>
<b>4.1.5 FUNÇÃO AUXILIAR – BUSCAR INCÓGNITA .....</b>	<b>15</b>
<b>4.1.6 FUNÇÃO AUXILIAR – TRANSFORMAR “CHAR” EM “DOUBLE” .....</b>	<b>16</b>
<b>4.1.7 FUNÇÃO AUXILIAR – CALCULAR OPERAÇÃO .....</b>	<b>16</b>
<b>5. PARTICIPAÇÕES.....</b>	<b>19</b>
<b>5.1 FILIPI SCALVIN DA COSTA .....</b>	<b>19</b>
<b>5.2 JOÃO PEDRO PRUDÊNCIO .....</b>	<b>19</b>
<b>6. CONCLUSÕES.....</b>	<b>20</b>
<b>7. REFERÊNCIAS.....</b>	<b>21</b>

## **1. OBJETIVOS**

O relatório que segue diz respeito aos procedimentos, códigos-fonte e demais situações abordadas na avaliação um (1) da disciplina de Estrutura de Dados do curso de Ciência da Computação. Tal documento tem caráter explicativo e elucidativo quanto aos quesitos avaliativos solicitados pelo professor, trazendo um conjunto de imagens para ilustrar o que foi solicitado no documento base, que contém as instruções desta atividade.

### 3. RESUMO

No vasto campo da resolução de expressões matemáticas, diversas notações desempenham um papel fundamental na simplificação e interpretação precisa dessas fórmulas matemáticas. Três notações notáveis que merecem destaque são a notação polonesa, a notação húngara e a notação polonesa inversa. Cada uma dessas notações tem suas próprias características e aplicações, contribuindo significativamente para a forma como expressões matemáticas são representadas, manipuladas e resolvidas.

A notação polonesa, também conhecida como notação prefixa, foi desenvolvida pelo matemático polonês “Jan Łukasiewicz” na década de 1920. Ela se destaca por sua ênfase na ordem dos operadores em uma expressão, colocando o operador antes dos operandos. Isso elimina qualquer ambiguidade na ordem de execução e torna a avaliação das expressões mais direta.

Por outro lado, a notação húngara, criada por “Charles Simonyi” durante seu tempo na Microsoft, é mais comumente associada à programação e não tanto à matemática. Essa notação envolve a incorporação de informações sobre o tipo de dados diretamente nos nomes das variáveis, tornando mais fácil para os programadores entenderem e manipularem variáveis em seus códigos.

No contexto da resolução de expressões matemáticas, a notação polonesa inversa, também conhecida como notação pós-fixa, oferece diversas vantagens. Ela se concentra em posicionar o operador após os operandos, o que resulta em uma representação mais clara das operações, tornando a interpretação das expressões matemáticas mais intuitiva. Além disso, a notação polonesa inversa elimina qualquer ambiguidade na ordem de execução, o que é particularmente útil em situações em que expressões complexas com múltiplas operações e expressões aninhadas precisam ser resolvidas de maneira eficaz.

Neste trabalho, o nosso objetivo é criar um algoritmo capaz de receber uma expressão algébrica, transformá-la na notação polonesa reversa desejada e calcular o seu valor total. Para isso, exploraremos em detalhes a notação polonesa inversa e sua aplicação na resolução de expressões matemáticas. Vamos discutir as vantagens e desafios relacionados a essa notação, além de apresentar a construção de uma

aplicação especializada que analisa e resolve expressões matemáticas utilizando a notação polonesa inversa como intermediária.

#### 4. DESENVOLVIMENTO

O desenvolvimento de uma solução eficiente para transformar uma expressão algébrica em notação polonesa inversa e, em seguida, calcular seu valor, representa um avanço significativo na manipulação e resolução de equações matemáticas. Essa abordagem oferece uma série de benefícios importantes no tratamento de expressões matemáticas, tornando o processo mais intuitivo e preciso.

Ao converter a expressão para notação polonesa inversa, a solução elimina a necessidade de considerar a ordem de operações e lidar com parênteses complexos. Isso simplifica substancialmente o processo de avaliação e interpretação da expressão. Além disso, a notação polonesa inversa reduz a ambiguidade, tornando claro o fluxo de operações a serem realizadas.

A capacidade de calcular o valor da expressão após a conversão é fundamental para a utilidade da solução. Afinal, a resolução das equações é o objetivo final, e a solução deve ser capaz de realizar essa tarefa com precisão e eficiência. Isso significa que os usuários podem obter resultados instantâneos e confiáveis para suas expressões matemáticas, independentemente da complexidade das fórmulas envolvidas.

Além disso, essa abordagem é especialmente benéfica para aplicações que envolvem cálculos matemáticos frequentes, como calculadoras, sistemas de processamento de dados, software de análise financeira e muito mais. A capacidade de transformar e resolver expressões rapidamente contribui para melhorar a produtividade, economizar tempo e reduzir erros em uma ampla gama de domínios.

Em resumo, a criação de uma solução que transforma expressões algébricas em notação polonesa inversa e, posteriormente, realiza a resolução dessas expressões, representa uma ferramenta poderosa com ampla aplicabilidade. Simplifica a interpretação e execução de equações complexas, resultando em maior precisão e eficiência em uma variedade de contextos matemáticos e científicos, bem como em aplicações que envolvem cálculos frequentes. Essa solução oferece benefícios tangíveis, economizando tempo, reduzindo erros e aprimorando a produtividade em várias áreas de atuação.

## 4.1 IMPLEMENTAÇÃO DO ALGORITMO


A função inicia seu processo ao receber a expressão como entrada. Ela procede mapeando cada caractere, aplicando uma condição específica que compara se é uma abertura ou fechamento, enquanto registra o número de ocorrências de cada tipo. Em última análise, a função verifica se o número de aberturas difere do número de fechamentos e, com base nessa comparação, retorna um valor booleano. Esse valor é crucial para a função chamadora, pois será usado para desencadear a exceção quando necessário, como exemplificado abaixo.

```
1  bool parentesesNaoForamFechados(const string& expressao) { // ou nao foram abertos
2      int aberturas = 0;
3      int fechamentos = 0;
4
5      for (char c : expressao) {
6          if (c == '(') {
7              aberturas++;
8          }
9          else if (c == ')') {
10             fechamentos++;
11         }
12     }
13
14     return aberturas != fechamentos;
15 }
```

```
1  if (parentesesNaoForamFechados(expressao)) {
2      cout << "\nQuantidade de parenteses invalida!!";
3      return 0;
4  };
```

Em uma sequência de passos, procedemos com o mapeamento da expressão, encaminhando individualmente cada caractere para uma fila específica, que denominamos de "Hungara".





```


1  for (char it : expressao) {
2      if (it != ' ')
3          inserir(hungara, it);
4  }

```

Em um fluxo sequencial de operações, avançamos para a próxima etapa, que consiste em iterar através da fila que foi criada anteriormente, denominada "Hungara". Essa fila contém os caracteres da expressão matemática após o mapeamento.

Durante essa iteração, utilizamos uma função específica de remoção da fila para obter o caractere que está atualmente sendo analisado. Essa função de remoção é responsável por retirar um elemento da fila e nos fornece o caractere para avaliação.

Uma vez que temos o caractere em mãos, prosseguimos com a validação do seu conteúdo. Isso significa que verificamos se o caractere atende a critérios específicos ou se está em conformidade com alguma regra pré-estabelecida para garantir que a expressão matemática seja processada corretamente.



```

1  while (remover(hungara, it)) {
2      char aux = NULL;
3      if (it == ')') {...}
4      else if (ehOperacao(it)) {...}
5      else if (ehOperando(it)) {...}
6  }

```

Inicialmente, realizamos uma validação para determinar se o caractere atual corresponde ao fechamento de um parêntese. Se essa condição for satisfeita, executamos uma ação específica. Nesse caso, removemos todos os operadores que estão armazenados na pilha de operadores até encontrarmos o correspondente parêntese de abertura.

```

1  if (it == ')') {
2      while (remover(operadores, aux) && aux != '(')
3          inserir(polonesa, aux);
4  }

```

Se o caractere atual for uma operação (utilizamos uma função auxiliar para verificar esse cenário), procedemos com a seguinte operação: retiramos todos os operadores que têm uma precedência menor do que a operação atualmente armazenada na variável de iteração e os inserimos na fila de notação polonesa inversa. Para esclarecer, como exemplo, se estivermos inserindo um operador de adição (+) na pilha, é necessário que antes retiremos todos os operadores de multiplicação (\*), divisão (/) e potenciação (^) que estão na pilha.

```

1  else if (ehOperacao(it)) {
2      while (remover(operadores, aux) && ehMenorPrecedente(it, aux)) {
3          inserir(polonesa, aux);
4          aux = NULL;
5      }
6      if (aux != NULL)
7          inserir(operadores, aux);
8      inserir(operadores, it);
9  }

```

#### 4.1.1 FUNÇÃO AUXILIAR – É OPERAÇÃO?

A chamada da função “ehOperacao” que verifica se um caractere dado é um operador matemático. Ela retorna true se o caractere for um operador (+, -, \*, / ou ^) e false caso contrário. A verificação é feita comparando o caractere com os códigos ASCII correspondentes a esses operadores.

```

1 bool ehOperacao(const char caractere) {
2     return caractere >= '(' && caractere <= '/' || caractere == '^';
3 }

```

#### 4.1.2 FUNÇÃO AUXILIAR - MENOR PRECEDENTE

A chamada da função “ehMenorPrecedente” que compara a precedência de dois operadores matemáticos. A função retorna “verdadeiro” se o primeiro operador (op1) tiver precedência menor do que o segundo operador (op2) e false caso contrário. A comparação de precedência é baseada nas regras padrão da matemática, onde algumas operações têm precedência sobre outras.

- Se op1 for adição (+) ou subtração (-), a função verifica se op2 é multiplicação (\*), divisão (/) ou potência (^). Se for, op1 é considerado de menor precedência, e a função retorna “verdadeiro, caso contrário, retorna false.
- Se op1 for multiplicação (\*) ou divisão (/), a função verifica se op2 é potência (^). Se for, op1 é considerado de menor precedência, e a função retorna “verdadeiro, caso contrário, retorna false.
- Se nenhuma das condições acima for atendida, a função retorna false, indicando que op1 não é de menor precedência em relação a op2.

Essa função é útil ao analisar expressões matemáticas para garantir que as operações sejam executadas na ordem correta de acordo com as regras de precedência.

```

1 bool ehMenorPrecedente(const char op1, const char op2) {
2     if (op1 == '+' || op1 == '-') {
3         return op2 == '*' || op2 == '/' || op2 == '^';
4     }
5     if (op1 == '*' || op1 == '/') {
6         return op2 == '^';
7     }
8     return false;
9 }

```

Retornando a fluxo geral do código, abordamos o último cenário, que envolve a validação do caractere como um operando. Realizamos essa validação com base no valor anterior. Se o caractere for confirmado como um operando, há uma exceção relevante a ser considerada: valores maiores do que 9 devem ser tratados como incógnitas. Caso não ocorra essa exceção, procedemos com a inserção do caractere na fila de notação polonesa inversa, utilizando a variável de iteração atual.

```
1  else if (ehOperando(it)) {
2      if (ehOperando(itAnterior)) {
3          cout << "\nOperandos menores que 0 ou maiores que 9 devem ser inseridos "
4              "como incognitas!";
5          return 0;
6      }
7      inserir(polonesa, it);
8  }
```

Quando retiramos todos os valores da fila "Hungara", transferimos todos os operadores que estão armazenados na pilha para a fila de notação polonesa inversa. É importante ressaltar que, nessa etapa, fazemos uso de uma fila auxiliar exclusivamente com o propósito de formatar e imprimir a expressão no formato Polonês.

```
1  cout << "\nExpressao polonesa inversa: ";
2  while (remover(polonesa, it)) {
3      inserir(polonesaAux, it);
4      cout << it;
5  }
6  cout << endl;
```

#### 4.1.3 FUNÇÃO AUXILIAR – É OPERANDO?

Esta função chamada “ehOperando” que verifica se um caractere dado é um operando válido em uma expressão matemática. A função retorna “verdadeiro” se o

caractere for um dígito ("0" a "9"), uma letra minúscula ("a" a "z") ou uma letra maiúscula ("A" a "Z"), que são os caracteres típicos usados para representar incógnitas, constantes ou variáveis em expressões matemáticas. Caso contrário, a função retorna "falso".

```
1 bool ehOperando(const char caractere) {  
2     return caractere >= '0' && caractere <= '9' ||  
3         caractere >= 'a' && caractere <= 'z' ||  
4         caractere >= 'A' && caractere <= 'Z';  
5 }  
6
```

Para concluir o processo, realizamos o cálculo da expressão utilizando a função auxiliar "calcularExpressao" com base na fila de notação polonesa inversa.

```
1 cout << "\nResultado da expressao: " << calcularExpressao(polonesaAux) << "\n";
```

#### 4.1.4 FUNÇÃO AUXILIAR – CALCULAR EXPRESSÃO

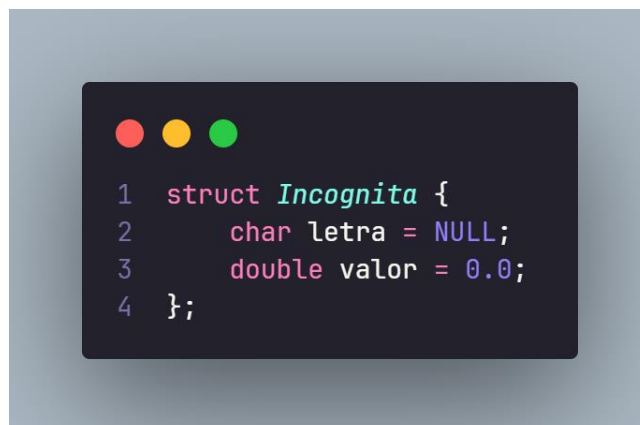
O código é responsável por calcular o valor de uma expressão matemática em notação polonesa inversa (RPN). Ele faz isso usando uma pilha chamada operandos para armazenar valores à medida que a expressão é avaliada. Aqui está como o código funciona:

- Ele percorre cada caractere da expressão em RPN usando um loop while.
- Para cada caractere, verifica se é um operando (um número ou uma variável). Se for um operando, converte o caractere em um número (caso seja um

número) ou busca o valor associado à variável (caso seja uma variável) e insere na pilha operandos.

- "Convertemos o caractere em um número (caso seja um dígito) usando a função 'char2double' porque precisamos representar operandos como valores numéricos na pilha 'operandos'."
- Se o caractere não for um operando, presume-se que seja um operador. Nesse caso, ele executa a operação correspondente com os operandos retirados da pilha operandos usando a função "calcularOperacao".
- Se ocorrer uma exceção durante o cálculo, o código imprime uma mensagem de exceção e retorna o resultado parcial até esse ponto.
- Após percorrer toda a expressão, o código retorna o resultado, que é o valor remanescente na pilha operandos.

A estrutura "Incognita" é utilizada para armazenar as associações entre letras (variáveis) e seus valores na expressão. A cada vez que uma incógnita é encontrada na expressão, o código busca seu valor na matriz "incógnitas" e a utiliza no cálculo. Essa estrutura é útil para avaliar expressões que contenham variáveis.

A imagem mostra um editor de código com um fundo escuro e uma borda arredondada. No topo, há três círculos decorativos: um vermelho, um amarelo e um verde. O código é escrito em uma fonte monoespaçada com coloração sintática. Ele define uma estrutura chamada 'Incognita' com dois membros: 'letra' do tipo 'char' inicializado com 'NULL' e 'valor' do tipo 'double' inicializado com '0.0'.

```
1 struct Incognita {  
2     char letra = NULL;  
3     double valor = 0.0;  
4 };
```

#### 4.1.5 FUNÇÃO AUXILIAR – BUSCAR INCÓGNITA

A função "buscarIncognita" é utilizada para encontrar e recuperar os valores associados às variáveis (incógnitas) presentes na expressão matemática em notação polonesa inversa (RPN). O motivo para usar essa função é lidar com as variáveis de forma apropriada durante a avaliação da expressão.

```

1  double buscarIncognita(Incognita vetor[52], const char incognita) {
2      int i = 0;
3      for (; i < 52 && vetor[i].letra != NULL; i++)
4          if (vetor[i].letra == incognita)
5              return vetor[i].valor;
6      cout << "\nInsira o valor da incognita " << incognita << ": ";
7      cin >> vetor[i].valor;
8
9      vetor[i].letra = incognita;
10     return vetor[i].valor;
11 }

```

#### 4.1.6 FUNÇÃO AUXILIAR – TRANSFORMAR “CHAR” EM “DOUBLE”

Esta função permite que o código reconheça os dígitos como números reais e os insira corretamente na pilha de operandos para posterior cálculo. Sem essa conversão, os dígitos seriam tratados apenas como caracteres simples e não participariam das operações matemáticas.

```

1  double char2double(const char c) {
2      char i = '0';
3      double valor = 0.0;
4      while (i != c && i != '9' + 1) {
5          i++;
6          valor++;
7      }
8      return i == c ? valor : -1;
9  }

```

#### 4.1.7 FUNÇÃO AUXILIAR – CALCULAR OPERAÇÃO

Esta função é responsável por realizar operações matemáticas com base em um operador (operação) fornecido em uma expressão em notação polonesa inversa (RPN). Ela segue estas etapas:

- Declara duas variáveis, “operando1” e “operando2”, para armazenar os operandos retirados da pilha.
- Verifica se é possível remover com sucesso dois operandos da pilha operandos. Caso contrário, imprime uma mensagem de erro e retorna false, indicando um erro de excesso de operadores.
- Usa um bloco “switch” para determinar qual operação matemática deve ser realizada com base no operador fornecido. As operações incluem potência, multiplicação, divisão, adição e subtração.
- Calcula o resultado da operação com os operandos “operando1” e “operando2” e armazena-o na variável resultado.
- Se o operador não corresponder a nenhum dos casos no “switch”, a função imprime uma mensagem de erro e retorna false, indicando um operador inválido.
- Se a operação for bem-sucedida, a função retorna “verdadeiro”, indicando que a operação foi realizada com sucesso e o resultado está disponível em resultado.

Em resumo, a função “calcularOperacao” é fundamental para a avaliação de expressões RPN, retirando operandos da pilha, realizando operações matemáticas e tratando erros quando necessário.



```
1  bool calcularOperacao(Pilha<double>& operandos, const char operacao, double& resultado) {
2      double operando1, operando2;
3      if (!remover(operandos, operando1) || !remover(operandos, operando2)) {
4          cout << "\nExcesso de operadores!";
5          return false;
6      }
7
8      switch (operacao) {
9          case '^':
10             resultado = calcularPotencia(operando1, operando2);
11             return true;
12          case '*':
13             resultado = operando1 * operando2;
14             return true;
15          case '/':
16             resultado = operando1 / operando2;
17             return true;
18          case '+':
19             resultado = operando1 + operando2;
20             return true;
21          case '-':
22             resultado = operando1 - operando2;
23             return true;
24      }
25
26      cout << "\nUso de operador invalido!";
27      return false;
28 }
```

## **5. PARTICIPAÇÕES**

Nesta seção, destacamos as contribuições desempenhadas pelos indivíduos deste projeto e que contribuíram para o sucesso dele.

### **5.1 FILIPI SCALVIN DA COSTA**

O algoritmo lógico em questão foi em grande parte desenvolvido pelo Filipi, que desempenhou um papel importante na criação e aprimoramento de várias partes do código. Ele contribuiu com suas ideias e expertise para encontrar as melhores soluções. Filipi debateu sobre diferentes partes do código e ajudou a escolher as abordagens mais eficazes para resolver os desafios.

### **5.2 JOÃO PEDRO PRUDÊNCIO**

João Pedro ficou responsável pelo desenvolvimento do “relatório técnico” do trabalho, evidenciando o que foi desenvolvido e a lógica utilizada em cada sessão do algoritmo. Juntamente a isto, esteve participando na criação de aprimoramento de diversas partes do código, desejando apresentar seus pontos de vista sob o tema.

## 6. CONCLUSÕES

A implementação bem-sucedida do algoritmo de notação polonesa inversa destaca a eficácia dessa abordagem na simplificação e otimização do processo de resolução de expressões matemáticas. A notação polonesa inversa, também conhecida como notação pós-fixa, oferece uma série de vantagens que a tornam uma ferramenta valiosa.

Uma das principais vantagens da notação polonesa inversa é a eliminação da ambiguidade na ordem de execução das operações. Isso torna a interpretação das expressões direta e precisa, reduzindo a necessidade de uso excessivo de parênteses para estabelecer a prioridade das operações.

Além disso, essa notação simplifica a avaliação de expressões, resultando em eficiência computacional e economia de recursos. Isso é particularmente valioso em situações em que o desempenho é crucial, como em calculadoras, sistemas de análise de dados e processamento de linguagem natural.

Em resumo, a implementação eficaz do algoritmo de notação polonesa inversa demonstra seu potencial para simplificar e acelerar o cálculo de expressões matemáticas em diversos contextos. Ela oferece clareza, eficiência e precisão, tornando-a uma escolha valiosa em uma ampla gama de aplicações que envolvem cálculos e processamento de dados.

## 7. REFERÊNCIAS

ROBERTO, Paulo. **Matemática Financeira: Princípios de Funcionamento da HP 12-C.**

Recuperado de

[http://www.umotimoempreendedor.com/Palestra/Arq/HP\\_12C\\_Paulo\\_Roberto.pdf](http://www.umotimoempreendedor.com/Palestra/Arq/HP_12C_Paulo_Roberto.pdf).

Acessado em: 04 set. 2023.

Freund, J. E., & Tomita, M. . (1978). **Comunicação científica e tecnológica: a Disseminação Seletiva de Informações.** *Revista De Biblioteconomia De Brasília*, 6(2), 155–170. Recuperado de

<https://periodicos.unb.br/index.php/rbbsb/article/view/29184>. Acessado em: 04 set. 2023

JÚNIOR, Walteno. (2016). **Pilhas e Filas.** Recuperado de

[http://waltenomartins.com.br/algdados\\_apostila\\_pilhafile.pdf](http://waltenomartins.com.br/algdados_apostila_pilhafile.pdf). Acessado em 04 set. 2023.

BORTOLOTTTO, Franco et al. (2004). **4ª Fase - Implementação de uma Calculadora de Expressões Infixas na Linguagem C++ Utilizando a Ferramenta Dev C++.** São Paulo. Recuperado de

<https://www.periodicos.unesc.net/ojs/index.php/sulcomp/article/view/2069>