Trabalho Prático 1
Resolvedor de Expressão Numérica

Lucas Albano Olive Cruz - 2022036209

Departamento de Ciência da Computação (DCC) - Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte - MG - Brasil

lucasalbano@dcc.ufmg.br

1. Introdução

O trabalho proposto consiste na implementação de um resolvedor de expressões numéricas para uma turma de matemática em uma escola de ensino fundamental. O resolvedor deve fornecer o resultado das expressões e ser capaz de converter as expressões para a forma usual chamada infixa e para a forma polonesa inversa chamada posfixa.

Para isso, deve ser implementada uma representação para a expressão usando TADs, além da utilização de algoritmos para a resolução e conversão das expressões. Também é um requisito a análise de complexidade de tempo e espaço dos procedimentos implementados. Por fim, é necessário que o programa possua tratamento de erros e tolerância a falhas.

2. Método

Como anteriormente descrito, serão utilizados TADs para representar a expressão e também para auxiliar na execução dos algoritmos de inserção e resolução das expressões. Neste tópico descreveremos essas estruturas, porque e como foram implementadas em termos de classes e funções, além da descrição dos algoritmos utilizados.

2.1. Estruturas de Dados

A estrutura de dados escolhida para a representação da expressão foi a estrutura de árvores, mais especificamente uma árvore sintática. A escolha se deve à facilidade de resolução e conversão para as duas notações uma vez que a expressão é representada desta forma.

Também foram implementadas dois tipos de pilhas, uma para armazenar nós da árvore e outra para armazenar valores float, ambas com alocação dinâmica de memória. Estas serão utilizadas para auxiliar no posicionamento correto dos nós na árvore e para resolver a expressão.

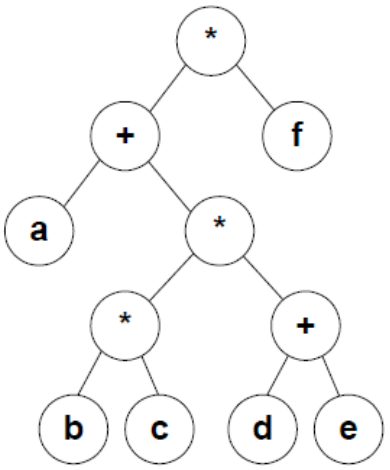
2.1.1. Árvore Sintática

A estrutura de dados árvore possui características que remetem a árvores reais com galhos que são representados por arestas que conectam vértices ou nós. Os nós mais

exteriores, que não possuem “filhos” são chamados de folhas. A raiz é o primeiro nó, ou vértice, de nossa árvore e a partir dele conectamos novos nós filhos que podem novamente ter novos nós conectados a eles, dessa forma criamos uma hierarquia de nós pais e filhos. Para nossa tarefa utilizaremos uma estrutura de árvore sintática.

A árvore sintática é na verdade uma árvore binária, ou seja, cada nó pai possui no máximo dois nós filhos. Dessa forma, as folhas da nossa árvore representam os números da nossa expressão e os nós internos as operações. A figura 1 ilustra essa definição.

Essa representação é extremamente útil pois usando diferentes algoritmos de caminhamento na árvore podemos reescrever a expressão com a notação infixa, posfixa ou resolvê-la.



.

Figura 1 - Representação da árvore sintática.

2.1.2. Pilhas

Uma pilha, como estudado ao longo da matéria, é uma estrutura com a característica que o último item a ser inserido é o primeiro a ser retirado, exatamente como em uma pilha de pratos ou livros. Essa característica será útil para os algoritmos de inserção e resolução das expressões.

Foi necessário a implementação de dois tipos de pilha. Uma para armazenar valores float e outra para armazenar nós de árvore. Possivelmente templates poderiam ser utilizados, mas por praticidade decidi implementar as estruturas como duas classes diferentes.

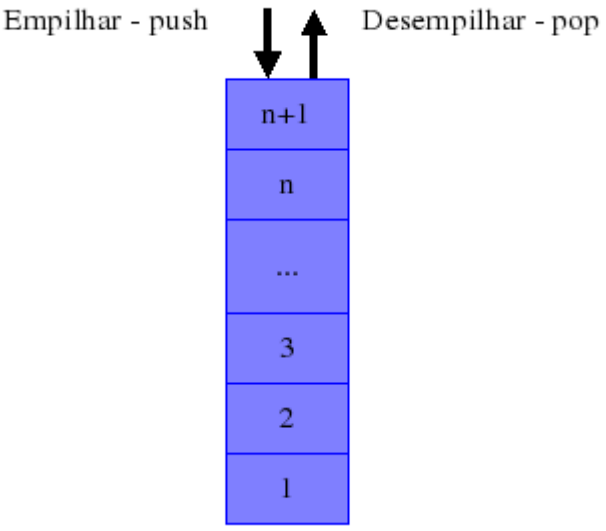


Figura 2 - Exemplo de uma pilha de inteiros.

2.2. Classes

Como descrito, teremos duas estruturas de dados implementadas em nosso programa com dois tipos diferentes de pilhas. Isso totaliza 8 classes, como descreveremos a seguir. Além destas, temos a classe resolvedora que controla a verificação, inserção e resolução das expressões.

2.2.1. TipoNo

Essa classe representa os nós através dos quais organizaremos nossa árvore. Ela possui um atributo item que guardará nossos operadores e operandos através de uma string. Além disso, possui dois ponteiros, um que aponta para o nó à esquerda e outro para o nó à direita.

Quando criamos um novo nó ele começa com uma string vazia e os dois ponteiros nulos. Também podemos criar nós passando uma string como parâmetro para ser atribuída ao item tal como podemos passar outro nó para ser copiado, tanto seu item como ponteiros.

Essa classe pode ter seus atributos privados acessados pela árvore, pela pilha de nós e pelo resolvedor.

2.2.2. ArvoreBinaria

É a classe que efetivamente monta e controla os nós formando uma árvore. Possui apenas um atributo TipoNo raiz que a partir dele podemos acessar todos os outros nós da árvore. A classe é capaz de inserir novos nós, tanto para expressões infixas como posfixas através de algoritmos que descreveremos a seguir. Também possui funções para caminhar através de nós de três formas diferentes: Pré-Ordem, Pós-Ordem e In-Ordem. Por fim, pode apagar todos os nós recursivamente e dizer se está vazia.

A classe Resolvedor pode acessar os atributos privados da árvore.

2.2.3. Pilha, PilhaEncadeada e node

Essas três classes juntas representam a estrutura pilha de nós e é dinamicamente alocada. Primeiramente a classe node é bastante semelhante a classe TipoNo, com a particularidade de possuir apenas um apontador para o próximo item ao invés de dois. Como item possui um ponteiro para um TipoNo. É inicializada com os dois ponteiros como nulo.

A classe pai Pilha possui apenas um atributo tamanho e funções para verificar esse tamanho ou se a pilha está vazia. Sua classe filha PilhaEncadeda herda todas as características de Pilha e as especializa. PilhaEncadeda possui um ponteiro para um node que representa o topo da pilha e funções para empilhar, desempilhar, fornecer o topo sem alterá-lo e limpar a pilha.

2.2.4. PilhaF, PilhaFloat, nodef

Essas três classes formam a estrutura de pilha de float e são idênticas as classes da pilha de nós. Sua única diferença é que o item ao invés de TipoNo é um float.

2.2.5. Resolvedor

A classe Resolvedor é a responsável por gerenciar a execução das operações requeridas, como ler a expressão, validar a expressão, converter para infixa, converter para posfixa e resolver a expressão.

Ela possui dois atributos sendo um a árvore que armazena a expressão e uma pilha para armazenar o resultado.

2.3. Algoritmos

Nessa seção irei descrever como funcionam os algoritmos de inserção, conversão e resolução do programa.

2.3.1. Inserir expressão Infixa

Quando ordenamos que o resolvedor leia uma expressão infixa antes de tudo ele verifica se a expressão é válida, depois limpa a árvore sintática para o caso de existir alguma outra expressão armazenada e então chama o método da classe árvore `InsereInfixa()` e passa a expressão como parâmetro.

Criamos então duas pilhas de nós vazias. Uma para armazenar os operadores e parênteses e uma para armazenar os operandos e os nós já montados.

Percorremos então toda a expressão e com uma variável auxiliar separamos a string usando os espaços para definir o fim de um caractere.

Caso o caractere seja um abre parênteses, o empilhamos na pilha de operadores.

Caso seja um operando o empilhamos na pilha de nós.

Caso seja um operador, precisamos verificar se o topo da pilha de operadores é outro operador com a mesma ou maior procedência. Se for o caso, desempilhamos o topo da pilha de operadores e dois nós da pilha de nós e empilhamos de volta na pilha de nós um nó com este operador e seus nós da direita e esquerda como o primeiro e segundo nó desempilhado respectivamente. Fazemos isso até que a condição falhe e só então inserimos o operador na pilha de operadores.

Caso o caractere seja um fecha parênteses, enquanto não encontrarmos um abre parênteses na pilha de operadores, desempilhamos o topo e outros dois nós da pilha de nós. Então empilhamos na pilha de nós um nó com este operador e seus nós da direita e esquerda como o primeiro e segundo nó desempilhado respectivamente, semelhante ao caso anterior. Assim que encontrarmos o abre parênteses correspondente o descartamos da pilha e também jogamos fora o fecha parênteses.

Depois de percorrer toda a expressão, somente um nó restará na pilha de nós, esse será a raiz da nossa árvore sintática.

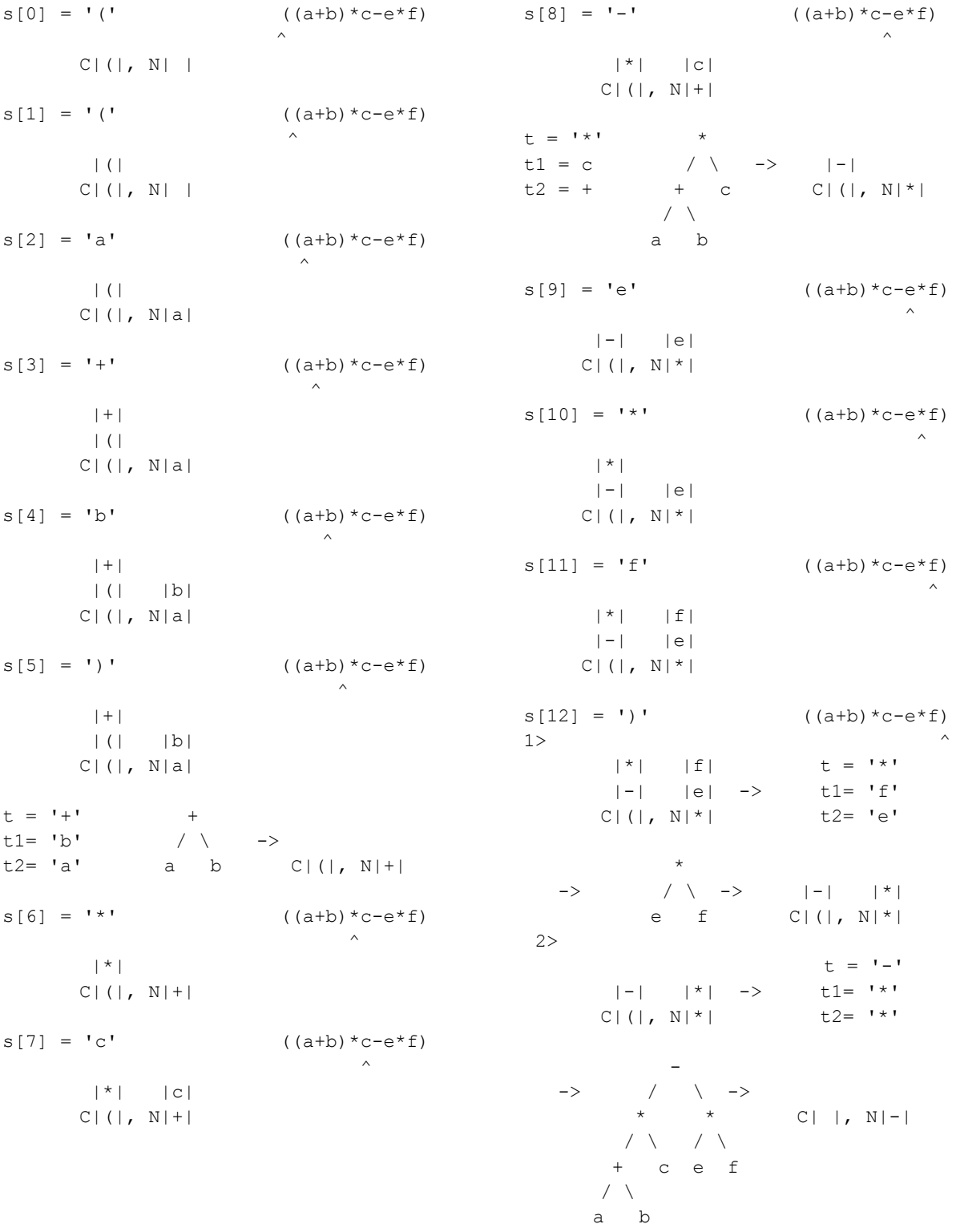
Vamos ver um exemplo de uso do algoritmo para a expressão ((a + b) * c - e * f).

Vamos representar a expressão como : s = ((a+b)*c-e*f)

Estou desconsiderando o passo de separar os caracteres utilizando os espaços.

Vamos usar C para denotar a pilha de operadores e N para denotar a pilha de nós.

Inicialmente ambas as pilhas estão vazias.



Por fim, tornamos (-) a raiz da árvore.

2.3.2. Inserir expressão Posfixa

Para inserir uma expressão posfixa o resolvedor segue passos semelhantes a expressão infixa. Primeiro ele verifica se a expressão é válida, limpa a árvore e chama o método da classe árvore `InserePosfixa()` e passa a expressão como parâmetro.

Criamos então uma pilha de nós auxiliar e percorremos toda a expressão separando os caracteres e os armazenando em uma variável através dos espaços.

Caso esse caractere seja um número o empilhamos na pilha auxiliar.

Caso seja um operador desempilhamos os dois últimos nós e os tornamos os nós a direita e esquerda do operador, então empilhamos o novo nó na pilha auxiliar.

Depois de percorrer toda a expressão teremos um único nó que representa a raiz de nossa árvore.

Pela menor complexidade desse algoritmo e semelhança no passo de criação de novos nós e inserção na pilha com o algoritmo anterior não irei demonstrar com um exemplo seu funcionamento.

2.3.3. Converter em Infixa

Para converter uma expressão em notação infixa desde que tenhamos uma árvore sintática é relativamente simples. Basta caminharmos na árvore usando um algoritmo de caminhamento in-ordem. Dessa forma, recursivamente vamos sempre indo para o nó à esquerda a partir da raiz da árvore, ainda, sempre que descemos um nível na árvore adicionamos um abre parênteses. Ao encontrar um ponteiro nulo voltamos um nível, imprimimos o caractere armazenado e então seguimos os mesmos passos no nó à direita. Ao voltar da recursão do nó à direita fechamos o parênteses. Ao terminar de percorrer a árvore teremos imprimido a expressão na notação infixa.

2.3.4. Converter em Posfixa

Para converter em notação posfixa seguimos passos semelhantes e ainda mais simples pois não precisamos dos parênteses. Basta percorrermos a árvore com um algoritmo de caminhamento pós-ordem. Dessa maneira, recursivamente vamos sempre indo para o nó à esquerda a partir da raiz, ao encontrar um ponteiro nulo voltamos um nível e vamos para o nó à direita no qual seguiremos os mesmos passos. Ao voltar da recursão à direita imprimimos o item. Ao terminar de percorrer a árvore teremos imprimido a expressão na notação pós-fixa.

2.3.5. Resolver a Expressão

Para resolvermos a expressão também percorremos a expressão com um algoritmo de caminhamento pós-ordem e utilizamos uma pilha de floats auxiliar. Seguimos os seguintes passos:

- Caso o caractere seja um número, o convertemos de string para float e o empilhamos.
- Caso seja um operador, desempilhamos os últimos dois números, executamos a operação e empilhamos o resultado de volta.

Após percorrer toda a árvore teremos apenas um elemento na pilha que será nosso resultado da expressão.

3. Análise de Complexidade

Nesse tópico analisaremos a complexidade de tempo e espaço dos algoritmos implementados.

3.1. Insere Infixa

O passo de percorrer toda a expressão possui complexidade de tempo $O(n)$ sendo n o tamanho da expressão. Os passos de comparações de caractere, empilhamento e desempilhamento são todos $O(1)$. Portanto, temos como complexidade total de tempo da operação $O(n)$.

Também temos complexidade de espaço igual a $O(n)$ pois o tamanho das duas pilhas utilizadas sempre será menor ou igual ao tamanho da expressão.

3.2. Insere Posfixa

Novamente percorremos toda a expressão com complexidade de tempo igual a $O(n)$ e empilhamos, desempilhamos e fazemos comparações com complexidade igual a $O(1)$. Portanto temos como complexidade de tempo total $O(n)$.

Também, a complexidade de espaço será igual a $O(n)$ pois nossa única pilha sempre terá tamanho menor ou igual a n, o tamanho da expressão.

3.3. Converte Infixa e Converte Posfixa

As duas operações de conversão são na verdade um algoritmo de caminhamento em árvore, sendo in-ordem para a forma infixa e pós-ordem para a forma posfixa. Portanto, os dois algoritmos possuem complexidade de tempo igual a $O(n)$ já que visitam cada nó da árvore sintática uma vez.

O algoritmo é implementado de forma recursiva, portanto, a complexidade de espaço também será $O(n)$ devido ao tamanho da pilha de recursão necessária para percorrer todos os nós.

3.4. Resolve Expressão

Para resolver a expressão o algoritmo segue os mesmos passos recursivos de um caminhamento pós-ordem, logo possui complexidade de tempo $O(n)$. A pilha de resultados sempre terá tamanho menor ou igual ao tamanho da expressão, portanto a complexidade de espaço é $O(n)$.

4. Estratégias de Robustez

Para proteger a integridade do programa as seguintes precauções foram tomadas:

- Caso o usuário insira um comando inexistente é retornado: `ERRO: OPERAÇÃO INVÁLIDA`
- Caso o usuário tente converter uma expressão ou resolvê-la antes de inserir a mesma é retornado: `ERRO: EXP NÃO EXISTE`
- Caso o usuário insira uma expressão inválida, como uma expressão que contenha caracteres inválidos, número incorreto de parênteses, número incorreto de operadores ou dois operandos seguidos em uma expressão infixa é retornado: `ERRO: EXP NÃO VÁLIDA`
- Caso o usuário tente resolver uma expressão que possua uma divisão por zero, o resultado é igualado a zero e é retornado: `ERRO: DIVISÃO POR ZERO`

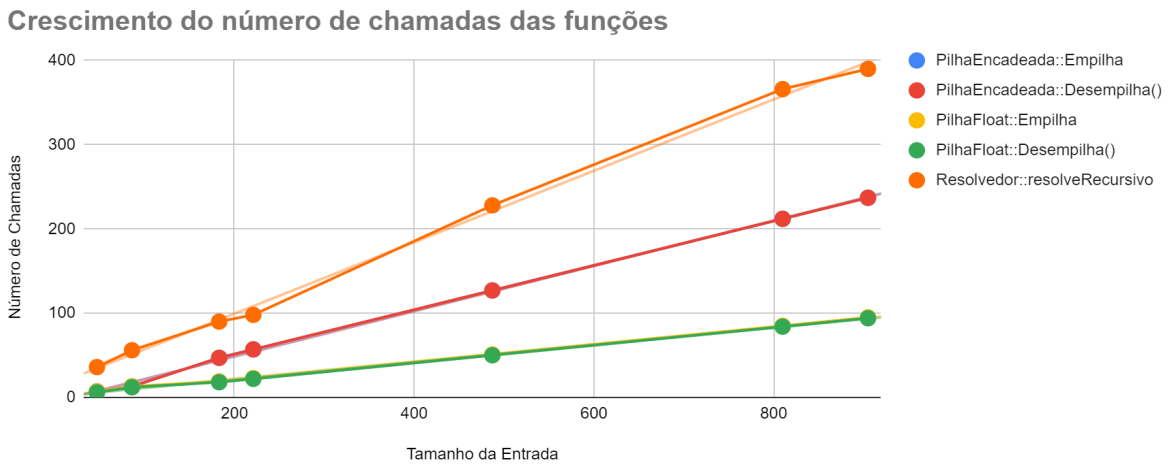
5. Análise Experimental

Na análise experimental do nosso programa irei utilizar a ferramenta gprof e analisar o crescimento da complexidade das funções conforme o crescimento do tamanho das entradas.

Dessa forma, para as principais funções do nosso programa temos a seguinte tabela que representa o número de chamadas de cada método:

Tamanho da Entrada	47	86	183	221	487	810	905
`PilhaEncadeada::Empilha(TipoNo*)`	7	13	47	57	127	212	237
`PilhaEncadeada::Desempilha()`	7	13	47	57	127	212	237
`PilhaFloat::Empilha(float)`	7	13	19	23	51	85	95
`PilhaFloat::Desempilha()`	6	12	18	22	50	84	94
`Resolvedor::resolveRecursivo(TipoNo*)`	36	56	90	98	228	366	390

Vamos representar isso em um gráfico para melhor visualização.



A partir do gráfico e da tabela podemos claramente concluir que o padrão de crescimento dos algoritmos implementados é linear , o que confirma nossa análise de complexidade assintótica.

6. Conclusões

Podemos então concluir que o programa desenvolvido atende aos requisitos propostos de forma consistente e objetiva, possui crescimento de tempo de execução e espaço necessário aceitáveis, além de robustez a erros do usuário.

Foi possível desenvolver diversos conhecimentos sobre estruturas de dados, em especial a estrutura de árvores. Também, foi possível praticar técnicas de análise de complexidade.

Os maiores desafios encontrados foram como montar a árvore sintática e desenvolver um algoritmo para isso, os quais através de pesquisas na internet consegui encontrar inspirações.

7. Bibliografia

CHAIMOWICZ, L. and PRATES, R. (2020). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

GeeksforGeeks, 2021. Program to convert Infix notation to Expression Tree. Disponível em: https://www.geeksforgeeks.org/program-to-convert-infix-notation-to-expression-tree/. Acesso em: 30 abr. 2023.

KHATRI, Jayanti. Expression Tree from Postfix. Youtube, 16 out. 2019. Disponível em: https://youtu.be/WHs-wSo33MM.

Apêndice A - Instruções para Compilação e Execução

Para compilar o arquivo primeiro descompacte o arquivo .zip e acesse a pasta raiz do projeto. Na pasta raiz execute o comando `make` para compilar. O executável será armazenado na pasta `bin` com o nome `programa`.

Para executar o programa acesse diretamente o programa com `./bin/programa` ou use o comando `make run`.

Ao executar o programa ele exibirá no terminal a lista de comandos aceitos. São eles:

- LER <tipo> <expressão> - Lê uma expressão e a armazena
    - Exemplo: LER POSFIXA 9.42141 7.32141 * 2.24654 +
- POSFIXA - Converte a expressão armazenada para a forma posfixa e a imprime
- INFIXA - Converte a expressão armazenada para a forma infixa e a imprime
- RESOLVE - Resolve a expressão armazenada e imprime o resultado
- SAIR - Encerra o programa

Instruções adicionais do utilitário makefile:

- `make clean` - Deleta os arquivos .o e o executável.
- `make teste` - Deleta os arquivos .o e o executável , em seguida, compila o programa e o executa.