UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO

Resolvedor de Expressão Numérica

Aluno: Lucas de Oliveira Araújo

Matrícula: 333

Belo Horizonte, MG
2023

Sumário

---

1 Introdução

Expressões matemáticas envolvem números interligados pelos sinais das operações fundamentais, ou seja, soma, subtração, multiplicação e divisão [1]. Essas expressões são usualmente resolvidas da esquerda para a direita, respeitando a ordem de precedência das operações.

Existem algumas notações para representar expressões numéricas. Nas escolas, comumente é ensinado a notação infixa, na qual os operadores são dispostos entre os números em que eles atuam (e.g. $(1 + 4) * (2 + 3) = 25$). Há também a notação polonesa reversa, amplamente conhecida como notação posfixa, a qual foi introduzida pelo filosofo e lógico australiano Charles Hamblin. Essa notação foi adaptada da notação polonesa (prefixa) inventada pelo matemático Jan Lukasiewicz, e seu intuito é dispensar a necessidade de parênteses para representar a ordem em que as operações devem ser feitas [2]. Para isso, os operadores são dispostos após os operandos em que eles atuaram (e.g. $1\ 4 + 2\ 3 + * = 25$).

O presente trabalho visa apresentar um programa que recebe uma expressão em notação infixa ou posfixa, resolve-a e retorna o resultado para o usuário, além de possibilitar que a expressão seja convertida entre as duas notações.

2 Método

2.1 Estruturas de dados

A representação interna de uma expressão é feita utilizando a estrutura de dados árvore binária. Escolhemos tal estrutura, pois a sua propriedade fundamental é armazenar valores conforme uma certa hierarquia. Portanto, isso possibilita que armazenemos nossas expressões de maneira em que a hierarquia das operações, isto é, a ordem de precedência, seja preservada.

Além disso, utilizamos outras duas estruturas de dados auxiliares: pilha e fila. Essas estruturas são utilizadas para a manipulação antes e após o armazenamento dos dados na árvore.

2.2 Tipos abstratos de dados

Nosso programa utiliza-se da boa prática de modularização, a fim de que a entendimento e a manutenção do código seja facilitada. Nesse sentido, dispomos de três classes principais: *ExpressionTreeCalculator*, *Parser* e *Converter*.

Outras classes são utilizadas para representar as estrutura de dados citadas anteriormente, além daquelas que são utilizadas no controle de exceções do nosso programa.

2.2.1 ExpressionTreeCalculator

A classe ExpressionTreeCalculator é o coração do programa. Ela é responsável por controlar a manipulação da nossa árvore binária, de maneira que seja possível armazenar uma expressão, resolvê-la e convertê-la, quando solicitado pelo usuário.

Os métodos principais desta classe são:

1. *postfix* e *infix*: Responsáveis por entregar a expressão em formato posfixa ou infixa, respectivamente, quando o usuário solicitar;

2. *showTree*: Esse método imprime o estado atual da árvore em um arquivo, no intuito de demonstrar como os dados foram organizados, o que facilita a análise de como a expressão foi armazenada internamente;

3. *Evaluation*: Esse método utiliza-se do caminhamento pós-ordem para percorrer a árvore e resolver a expressão que ela representa. O valor resultante é retornado para o usuário;

4. *storeExpression*: Responsável por armazenar uma expressão na árvore binária. A expressão armazenada na árvore segue o formato de uma expressão posfixa, ou seja, antes de armazená-la de fato, caso seja uma expressão em formato infixa, ela será convertida para posfixa e só então armazenada na árvore. A conversão de infixa para posfixa foi inspirada no algoritmo Shunting Yard, proposto por Edsger Dijkstra [3, 4].

O algoritmo Shunting Yard é um tanto elegante. Para converter uma expressão infixa para posfixa, precisamos de uma pilha e uma fila. A pilha é utilizada como um armazenamento de dados temporário, e a fila é a estrutura de dados que representará a nossa expressão convertida.

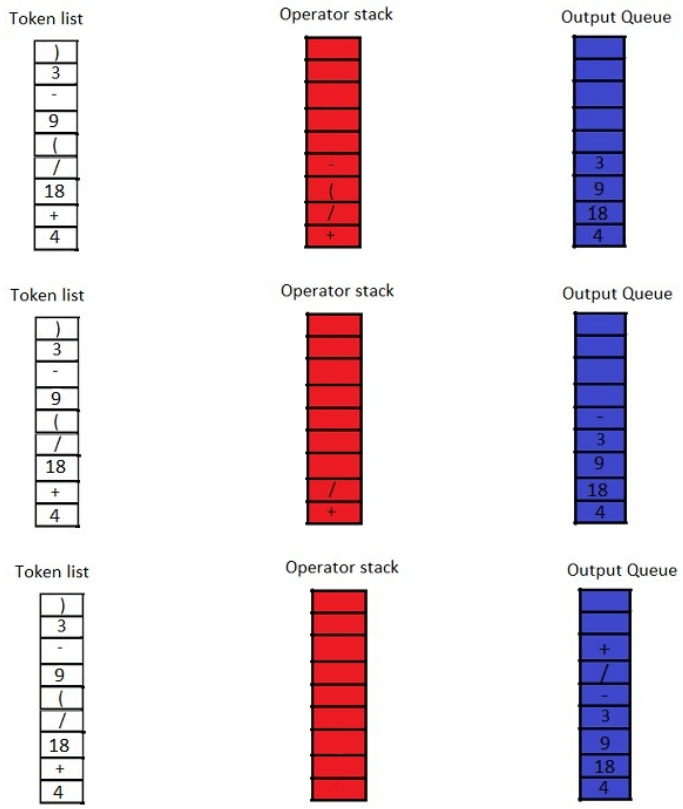

Figura 1: Shunting Yard em operação

Por exemplo, suponha que temos a expressão $4 + 18/(9 - 3)$ representada em uma string qualquer, conforme figura 1. Para a conversão, percorremos cada operador ou operando da esquerda para a direita. Quando encontrarmos um número, o direcionamos diretamente para a fila de saída (azul). No caso de encontrarmos um operador, o direcionamos para a pilha temporária (vermelho), e antes de o empilharmos fazemos a seguinte verificação:

1. A precedência do operador atual é menor ou igual a precedência do operador que está no topo da pilha

- Nessa caso, vamos desempilhando os operadores que estão na pilha e direcionando para a fila de saída até encontrarmos um operador com precedência menor do que o operador atual. Quando o encontrarmos, empilhamos o operador atual.

2. Se a precedência do operador atual for maior do que a precedência do operador que está no topo da pilha, então simplesmente empilhamos o operador.

No caso especial de o token atual ser a abertura de parênteses, então tudo que está na nossa string até o fechamento desse parêntese terá precedência maior. Nessa caso, empilhamos o parêntese aberto e vamos enfileirar os números na ordem em que eles aparecem na string e empilhar os operadores também na ordem em que eles aparecem na string até encontrar o fechamento desse parêntese. Quando isso ocorrer, vamos desempilhar todos os operadores e enviar para a fila até encontrar o parêntese aberto que empilhamos. Encontrado o parêntese, podemos desempilhá-lo e descartá-lo junto com o seu fechamento, e continuar com o fluxo do algoritmo.

Quando chegarmos ao fim da string, poderá restar alguns operadores na pilha temporária. Estes podem ser desempilhados e enviados à fila. Essas manipulações garantiram que a precedência das operações da expressão original será preservada. No final de todo esse trâmite, nossa fila representará a expressão no formato posfixa.

Portanto, fica evidente que a classe ExpressionTreeCalculator, munida de tais funções, controla todo o fluxo de operação do programa e como ele interage com o usuário.

2.2.2 Parser

O Parser é responsável pela validação dos dados que são disponibilizados pelo usuário. Essa classe contém apenas métodos estáticos, uma vez que o seu objetivo é o de apenas realizar validações.

As duas principais validações são a de se a expressão está representada na forma infixa ou posfixa, além de se é, em ambos os casos, válida.

2.2.3 Converter

A classe converter é responsável pelas conversões necessárias de um tipo de dado para outro. Para que isso ocorra acertadamente, ela trabalha em cima das validações da classe Parser. Dessa maneira, uma conversão só é realizada após a validação dos dados, o que evita as tentativas de conversão desregradas.

A classe converter também dispõe-se somente de métodos estáticos, uma vez que os dados são convertidos e entregues diretamente ao atributo ou método invocador, não sendo necessário qualquer armazenamento interno na classe.

Portanto, a classe converter é essencial para o bom funcionamento do programa, uma vez que, ao trabalhar de forma coerente, evita erros indesejados.

3 Análise de Complexidade

3.1 Complexidade de tempo

Quando o programa recebe uma expressão, seja em notação infixa ou em notação posfixa, sua primeira ação é verificar a validade desta, isto é, analisar se a sua estrutura é coerente e se não há quaisquer caracteres inválidos.

No que diz respeito aos caracteres inválidos, ou seja, todos aqueles que não são utilizados na construção de uma expressão válida, temos duas funções especiais na classe Parser:

1. *isNumber*: Percorre uma string no intuito de validar se ela representa um número válido;
2. *isValidOperator*: Verifica se um char representa um dos quatro operadores aritméticos (*, /, +, -).

isNumber tem complexidade linear, isto é, $O(n)$, uma vez que é necessário percorrer toda a string em busca de algum caractere inválido, além de realizar algumas verificações adicionais, por exemplo, verificar se não há dois separadores decimais nessa string. De qualquer maneira, as expressões geralmente não contém números compostos por dezenas de dígitos ou são formadas por milhares de números, o que naturalmente torna a complexidade de isNumber pouco relevante.

isValidOperador, por outro lado, tem complexidade constante, isto é, $O(1)$, pois ele apenas verifica um único char, não sendo necessário nenhum tipo de iteração.

Utilizando-se das duas pequenas funções citadas acima, os métodos que validam a estrutura intrínseca de uma expressão são *InfixIsValid* e *PostfixIsValid*. Esses dois métodos têm complexidade $O(n)$, onde n é a quantidade de operadores e operandos que há em uma expressão (em uma expressão em notação infixa, havendo parênteses, estes também são contabilizados). A complexidade linear se dá pelo fato de ser necessário analisar cada um dos tokens, isto é, cada subsequência de string que forma a expressão.

Ademais, os métodos de conversão entre uma notação e outra também apresentam complexidade $O(n)$ no pior caso. Isso se dá pelo fato de os métodos de conversão também percorrem a expressão, mas, em vez de verificar sua validade, apenas manipulam seus tokens sequencialmente, movendo-os para as estruturas de dados necessárias, conforme exposto na explicação do algoritmo Shunting Yard.

3.2 Complexidade de espaço

No tocante ao espaço de memória utilizado durante a execução do programa, citemos a classe ExpressionTreeCalculator, que é responsável por armazenar a expressão. Para o armazenamento, a referida classe toma-se do método *storeExpression*. Este utiliza-se da alocação dinâmica da memória para construir os nós da árvore. Como a quantidade de nós depende diretamente da quantidade de operadores e operandos da expressão, essa função tem complexidade de espaço $O(n)$, onde n é a quantidade de tokens utilizados para representar a expressão.

Quando uma nova expressão válida é disponibilizada pelo usuário, a árvore que representa a expressão atual é destruída e a nova expressão é armazenada. Dessa maneira, os nós da árvore antiga são desalocados, de maneira que a memória seja liberada e o programa não tenha um custo de espaço que ultrapasse o linear.

4 Estratégias de Robustez

A estratégia de robustez empregada foi o controle de exceções, além dos testes implementados utilizando a biblioteca DOCTEST[1].

O controle de exceções foi implementado em múltiplas partes do programa. Para cada uma das estruturas de dados utilizadas, existe um conjunto de exceções que são lançadas e tratadas, conforme

[1]DOCTEST pode ser encontrado no github: `https://github.com/doctest/doctest`

o necessário. Dessa maneira, erros malquistos são devidamente tratados, de forma que a experiência do usuário não seja desagradavelmente afetada.

Os testes foram implementados considerando o mais diversos tipos de cenários. Da mesma maneira que as exceções, para cada estrutura de dados implementada, há também um conjunto de testes realizados no intuito de confirmar a resiliência de tais estruturas.

Ademais, para as três principais classes do programa, a saber, ExpressionTreeCalculator, Parser e Converter, também foram implementados um conjunto de testes multifocais, visto que tais funções estão no cerne de todo o funcionamento do código. Dada essa importância, tais classes não poderiam ser utilizadas sem antes a confirmação de que elas trabalhariam conforme o esperado.

5 Análise Experimental

Dada a importância da classe expressionTreeCalculator para o funcionamento do programa como um todo, decidimos direcionar nossos experimentos para ela.

O experimento consistiu em verificar o tempo decorrido na execução dos quatro principais métodos dessa classe, a saber, os métodos storeExpression, infix, postfix e evaluation. A escolha foi feita pensando no fluxo de trabalho habitual de um possível usuário: primeiro ele vai disponibilizar a expressão, talvez convertê-la entre uma notação e outra, e por fim resolvê-la. Portanto, com o intuito de verificar o tempo médio necessário para que esse fluxo de comandos seja executado, geramos diversas expressões, com tamanhos que variavam entre 10 e 1000 tokens. O resultado final está exposto no gráfico da figura 2.

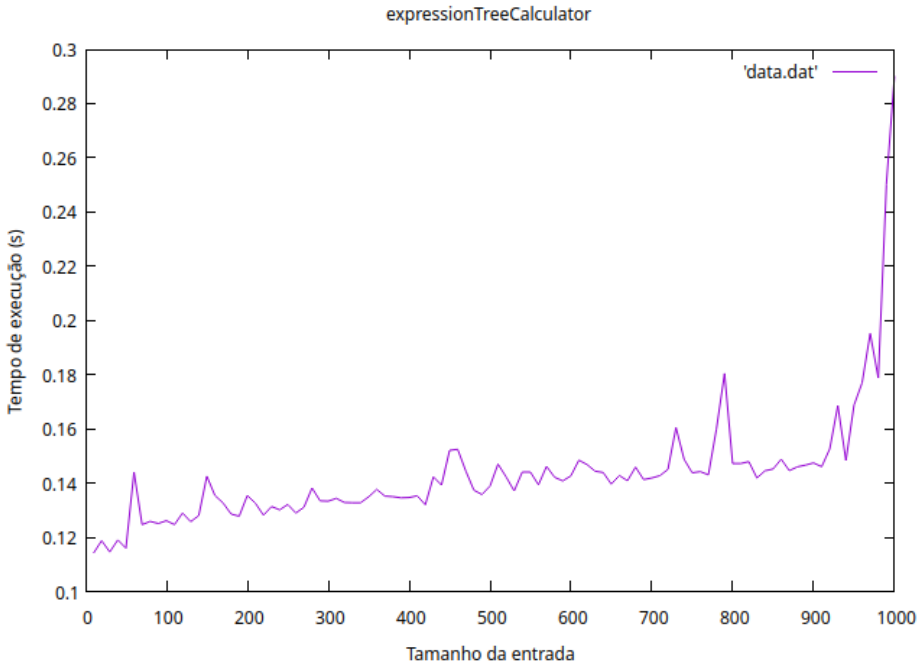

Figura 2: Comportamento da classe expressionTreeCalculator

Considerando os limites estabelecidos na especificação do programa (tamanho máximo de 1000 tokens por expressão), é possível notar que o tempo necessário para que esse fluxo de comandos seja executado aumenta de acordo com a entrada, o que condiz com a análise de complexidade

de tempo realizada anteriormente. Além do mais, percebê-se ainda que esse aumento não é tão ascendente como se poderia imaginar.

Portanto, é possível concluir que o programa não é tão sensível ao tamanho da entrada, apresentando bom comportamento em todo o intervalo de possíveis tamanhos de expressões.

6 Conclusões

Nesse trabalho foi proposto um programa que recebe e manipula expressões na notação infixa ou posfixa. Tal programa utiliza-se das estruturas de dados fila, pilha e árvore binária para manipular e armazenar as expressões, além de dispor-se de um ferramental completo no tocante a tratativa de possíveis erros.

Por fim, a construção do referido programa possibilitou um amplo aprendizado sobre a manipulação das estruturas de dados citadas, além de reforçar os conceitos básicos de programação orientada a objetos. Não menos importante, a implementação desse programa exigiu o estudo de algoritmos que possibilitassem a conversão entre as notações, sendo o já citado Shunting Yard o escolhido para a conversão de uma expressão infixa para posfixa. A partir dele, todo o processo de manipulação dos dados surgiu naturalmente, uma vez que a notação posfixa é um tanto mais simples de se trabalhar.

Referências

[1] Jose Carlos Admo Lacerda. *Praticando a Aritmética*. Editora XYZ, 2007. ISBN: 85-9037-951-5.

[2] *What is Reverse Polish Notation?* 2023. URL: `https://www.calculator.org/articles/Reverse_Polish_Notation.html` (acesso em 16/04/2023).

[3] Edsger Dijkstra. "Algol 60 translation : An Algol 60 translator for the X1 and making a translator for Algol 60". Jan. de 1961.

[4] *Shunting Yard Algorithm*. 2023. URL: `https://brilliant.org/wiki/shunting-yard-algorithm/` (acesso em 16/04/2023).

Apêndice

Instruções para compilação e execução

Uma vez no diretório raiz do programa, a compilação poderá ser realizada por meio do comando `make build`.

Após a compilação, o código poderá ser executado de duas maneiras. Se o intuito é passar os comandos através de um arquivo de texto, utilize `make run` seguido do operador de direcionamento dos sistemas Unix e do nome do arquivo, conforme ilustrado na figura 3. Você também pode passar vários arquivos de texto em sequência para o programa. Para isso, concatene um arquivo seguido do operador de direcionamento e do próximo arquivo. Ou então, tendo todos os arquivos de texto em uma determinada pasta, digamos, na pasta `entradas`, você poderá utilizar o comando `make run < entradas/*`. Assim, o programa receberá todas as entradas de todos os arquivos localizados nessa pasta.

Figura 3: Passando um arquivo com os comandos para o programa

Por outro lado, se o intuito é utilizar o programa interativamente, recomendamos que você utilize o comando `make run_interactive`. A partir disso o programa será aberto e aguardará pelos comandos. Digite `HELP` e pressione `enter` para visualizar a lista completa de comandos.

OBS.: O comando `make run`, sem o operador de direcionamento e os arquivos, também funciona de forma interativa, mas ele não foi configurado para esse propósito, logo, algumas opções como a `EXIT` podem não funcionar.

Ademais, os testes e o valgrind podem ser executados via os comandos `make tests` e `make valgrind`, respectivamente.