Calculadora 1.0

Vamos desenvolver um mecanismo de resolução de operações básicas usando a técnica de notação polonesa para resolver a equação

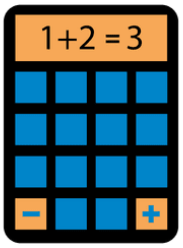
Receber a expressão em Números (Ex: 10+20*2)

Inicialmente, vamos criar um método de nome ***resolveExpressao*** que receberá como parâmetro uma ***String*** contendo a expressão a ser resolvida. Vou chamar essa expressão de expressão Original. Esse método vai transformar a expressão original em uma expressão polonesa invertida (veremos mais adiante) e vai resolvê-la usando estruturas de pilha e fila (que veremos mais adiante também) devolvendo como resultado, um valor do tipo ***double***, que é o padrão para números em ponto flutuante.

A assinatura do método vai ficar assim:

```
public static double resolveExpressao(String expOriginal)
                                                throws Exception
```

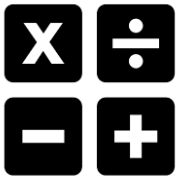
Neste método teremos a chamada dos métodos privativos que resolverão a nossa expressão.



Não vamos nos preocupar agora com a maneira de obter a expressão original. No final da nossa construção, podemos montar uma classe visual pra fazer o *front-end* da aplicação, cada um à sua maneira e seguindo a customização que desejar!

Para resolver a expressão, vamos precisar extrair dela os valores numéricos e armazená-los em uma estrutura que nos permita recuperá-los na sequência em que eles apareceram. Perceba que uma estrutura de Fila cai muito bem aqui! Uma lista também nos ajudaria, porém, como este ponto ainda não compõe os nossos conhecimentos já adquiridos, vamos de Fila!

Depois de extrair os valores e guardá-los em uma estrutura, vamos trabalhar a expressão para realocar as operações de forma que nos permita recuperar sempre dois valores e em seguida, a operação a ser aplicada a eles. Para que isso seja possível, precisaremos de uma versão da expressão que não contenha mais os valores expressos em números e sim, que tenha letras no lugar dos números. Sobre essa expressão, agora em letras, vamos aplicar a notação polonesa. A notação polonesa vai impor a prioridade das operações, resolvendo primeiro as operações matematicamente prioritárias ou as sub-expressões de uma expressão maior. Uma PILHA Vai nos ajudar nessa tarefa de criar a expressão em notação polonesa!



Depois de montadas as estruturas, vamos apenas aplicar a lógicas de resolução da expressão! Simples, não é mesmo!!

Agora vamos por a mão na massa pra resolver este problema em pequenos passos.

Veja como fica o nosso primeiro método e o único que será público, ou seja, o único que será acessível a partir da aplicação final.

```
public static double resolveExpressao(String expOriginal)
                                                    throws Exception
{
      Fila      numeros        =    extraiNumeros(expOriginal);
      String    expEmLetra     =    transformaLetras (expOriginal);
      String    expPolonesa    =    montaPolonesa(expEmLetra);
      return    resolvePolonesa(numeros, expPolonesa);
}
```

Vejamos agora as instruções para desenvolver cada parte do projeto.

Extrair os números e coloca-los em uma FILA

- Fazer a leitura da expressão recebida como parâmetro, caracter a caracter e, para cada um deles:
  - Sempre que encontrar um dígito numérico, guardá-lo em uma String temporária para montar um número (concatenando os dígitos). Você pode verificar se um caractere é um dos caracteres que representa um dígito numérico fazendo o seguinte teste

```
if ("0123456789".indexOf(caracter) >-1) {
    // caractere é um dígito numérico
}
```

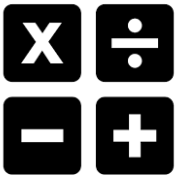
- Sempre que encontrar algo que não seja um dígito numérico, se tiver algo na String que monta o número, converter o número montado na String para double e mandar ele para a Fila. Limpar a String do número.

Gerar expressão em Letras

- Fazer a leitura da expressão recebida como parâmetro, caracter a caracter e, para cada um deles:
  - Sempre que encontrar um dígito numérico, indicar em uma variável booleana que temos um número. Você pode verificar se um caractere é um dos caracteres que representa um dígito numérico fazendo o seguinte teste

```
if ("0123456789".indexOf(caracter) >-1) {
    // caractere é um dígito numérico
}
```

- Sempre que encontrar algo que não seja um dígito numérico, se o booleano indicar que estávamos em um número, coloque uma nova letra na String final. Indicar no booleano que não estamos mais em um número. Colocar o caracter da vez na String final.

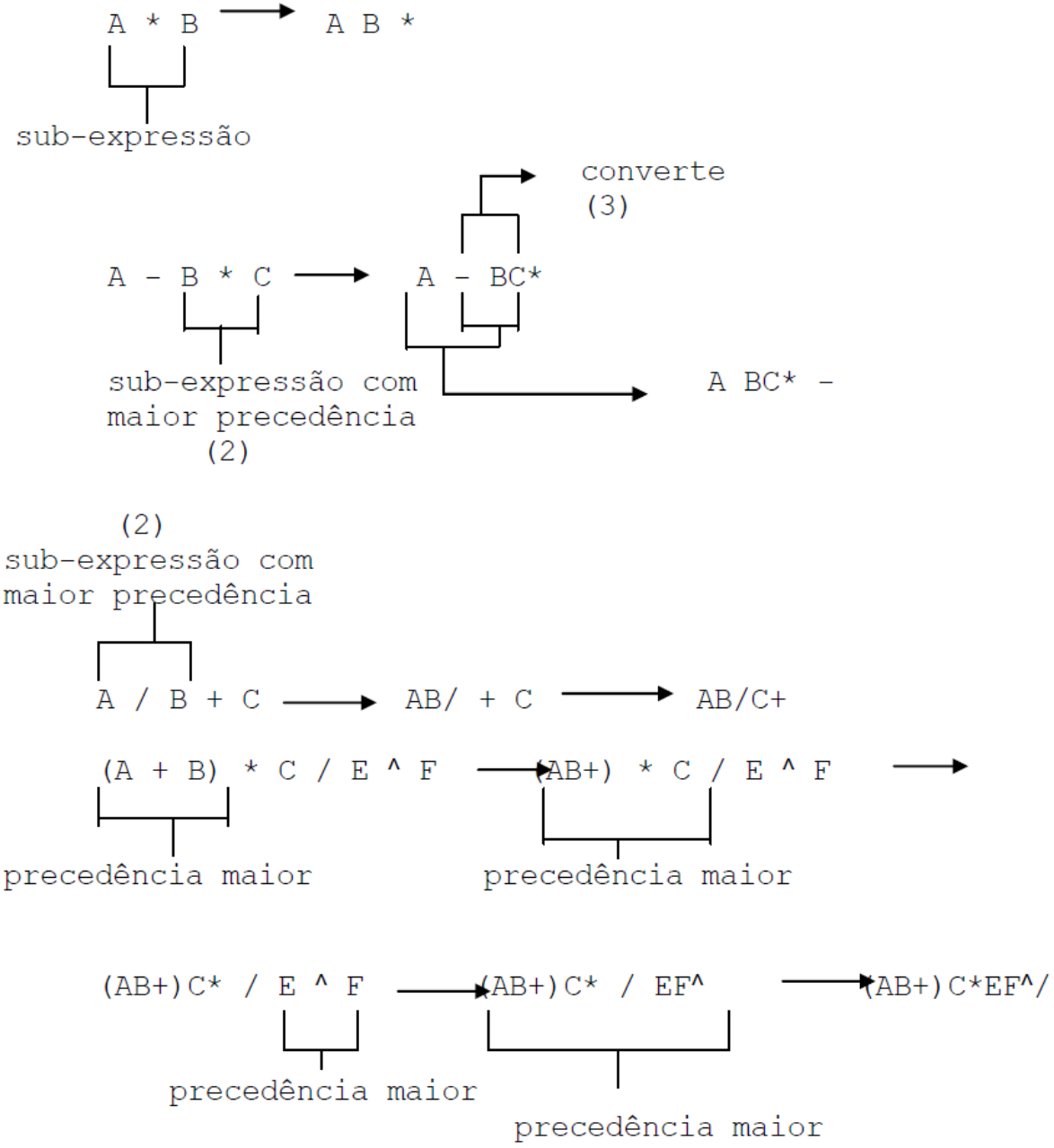


Gerar expressão pós-fixa

Para desenvolvermos um algoritmo de conversão de notação infixa para pós-fixa, observemos os passos a seguir:

1. verificar na sequência as sub-expressões que a formam
2. identificar a sub-expressão com maior precedência que ainda não tenha sido convertida; será algo do tipo A | B, onde A e B são operandos e | um operador binário.
3. Efetuar a conversão, colocando o operador após os dois operandos : AB|. Esse resultado passa a ser visto agora como um único operando.
4. Repetir as operações 2 e 3 até que se tenha terminado a conversão.

Exemplo:



Por fim, retiramos os parênteses e a expressão resultante é AB+C*EF^/.

Os parênteses não são necessários, pois agora os operadores aparecem na ordem de execução do cálculo, não sendo mais preciso mudar essa ordem.

Quando dois operadores de mesma precedência aparecerem em sequência, deve-se converter primeiro a sub-expressão mais à esquerda, já que esta apareceu antes que a outra (portanto, tem precedência sobre a outra). No entanto, quando os operadores em sequência forem , tem maior precedência a sub-expressão à direita. Exemplos:

```
A + B + C   ⟶  AB+ + C  ⟶  AB+C+
A ^ B ^ C   ⟶  A ^ BC^   ⟶  ABC^^

  A / B ^ C                    ABC^/
  A * B * C                    AB*C*
(A + B) / C * D              AB+C/D*
(A + B) / (C * D)            AB+CD*/
A + B ^ C / (D ^ E * F) / G / H    ABC^DE^F*/G/H/+
    A - B / (C * D ^ E)            ABCDE^*/-

((A-B) / C - (D+E)) ^ F ^ G  ⟶( AB- / C - DE+) ^ FG^   ⟶
( AB-C/ - DE+) FG^^   ⟶AB-C/DE+-FG^^
```

Algoritmo de conversão

(A ^ B * C - D + E / F / (G + H))

4 3 2 1 1 2 2 4 1 00

Os números abaixo da expressão indicam a prioridade (precedência) de cada operador.

Como se nota nos exemplos, a ordem de aparecimento dos operandos em ambas as sequências é a mesma. Logo, ao se fazer o algorítmo de conversão, deve-se escrever os operandos na sequência pósfixa à medida que são lidos da sequência infixa.

E os operadores? Para eles, devemos posfixar primeiro os mais prioritários; devemos ir guardando-os em uma pilha até encontrar um outro com prioridade maior no topo da pilha. Quando isso acontecer, devemos

colocar na sequência pós-fixa os operadores guardados (desempilhando-os), até o topo chegar a um operador de menor precedência que aquele operador lido, ou a pilha ficar vazia.

A regra, portanto, é guardar os operadores de maior precedência na pilha, até aparecer um de menor precedência que o topo, quando então deve-se descarregar a pilha até aparecer um operador de menor precedência que o lido, que então deve ser empilhado para uso futuro.

Assim, para a expressão acima, teremos :

Entrada Infixa	Pilha	Sequência Pósfixa
(A^B*C-D+E/F/(G+H))	(	
A^B*C-D+E/E/(G+H))	(	A
^B*C-D+E/F/(G+H))	(^	A
B*C-D+E/F/(G+H))	(^	AB
*C-D+E/F/(G+H))	(*	AB^
C-D+E/F/(G+H))	(*	AB^C
-D+E/F/(G+H))	(-	AB^C*
D+E/F/(G+H))	(-	AB^C*D
+E/F/(G+H))	(+	AB^C*D-
E/F/(G+H))	(+	AB^C*D-E
/F/(G+H))	(+/	AB^C*D-E
F/(G+H))	(+/	AB^*D-EF
/(G+H))	(+/	AB^C*D-EF/
(G+H))	(+/(	AB^C*D-EF/
G+H))	(+/(	AB^C*D-EF/G
+H))	(+/(+	AB^C*D-EF/G
H))	(+/(+	AB^C*D-EF/GH
))	(+/(+	AB^C*D-EF/GH+
)	(+/	AB^C*D-EF/GH+/+

Para implementarmos o algoritmo, vamos precisar definir bem as prioridades dos operadores. Para isso, pode-se montar uma tabela de precedência, como a que se segue; ela será usada para determinar se o operador lido deve ou não ser desempilhado:

		Simbolo pego da Sequência						
		(	^	*	/	+	-	)
Símbolo que está no topo da Pilha	(	F	F	F	F	F	F	T
	^	F	T	T	T	T	T	T
	*	F	F	T	T	T	T	T
	/	F	F	T	T	T	T	T
	+	F	F	F	F	T	T	T
	-	F	F	F	F	T	T	T
	)	F	F	F	F	F	F	F

Por exemplo, se pegarmos da sequência um símbolo igual a '' e se houver no topo da pilha um '(', este não tem precedência sobre o '' e, portanto, o '' deve ser empilhado.

Por outro lado, se pegarmos da sequência o símbolo '(' e se houver qualquer outro símbolo no topo da pilha, o símbolo lido deverá ser empilhado, pois não tem precedência sobre nenhum outro.

No entanto, se pegarmos da sequência '*' e se houver um '/' no topo da pilha, embora na matemática eles tenham a mesma precedência, o que apareceu primeiro deve ser desempilhado e colocado na sequência pósfixa; como o '/' teria aparecido antes na sequência infixa ele teria prioridade sobre o '*' e deverá ser desempilhado para aparecer na sequência pósfixa antes do '*', que, por sua vez, será empilhado antes de um elemento com precedência menor.

Resolver usando a pilha e a fila

De posse da lista de valores e da expressão polonesa, vamos percorrer a expressão e, com a ajuda de uma pilha, vamos resolvendo as operações uma após a outra, em ordem de aparição na expressão. Para cada operação, vamos simplesmente pegar dois valores e aplicá-los à operação da vez. Os dois primeiros valores seção retirados da estrutura de armazenamento dos valores e colocados na pilha resultante e, ao final, teremos o resultado armazenado na pilha.

Vejamos então que, a cada operando da expressão pós-fixa (ou seja, um valor), faremos um empilhamento. Quando aparecer um operador, desempilharemos os dois últimos elementos colocados na pilha de operandos, e aplicaremos a eles a operação correspondente ao operador. O resultado desta operação será empilhado, e o processo continua até que se chegue ao final da sequência pós-fixa. Nesse momento, haverá apenas um elemento na pilha, que será o resultado da expressão.

Como exemplo, suponha que temos a expressão abaixo:

AB-C/DE+-FG^^

Os valores dos operandos são os seguintes:

A = 23 B = 7 C = 8 D = 4 E = 2 F = 2 G = 2

O processo de cálculo vem descrito abaixo

Sequência pós-fixa	Pilha	
AB-C/DE+-FG^^	vazia	{ empilha valor de A }
B-C/DE+-FG^^	23	{ empilha valor de B }
-C/DE+-FG^^	23 7	{ calcula valor da subexpressão }
C/DE+-FG^^	16	{ empilha valor de C }
/DE+-FG^^	16 8	{ calcula valor da subexpressão }
DE+-FG^^	2	{ empilha valor de D }
E+-FG^^	2 4	{ empilha valor de E }
+-FG^^	2 4 2	{ calcula valor da subexpressão }
-FG^^	2 6	{ calcula valor da subexpressão }
FG^^	-4 2	{ empilha valor de F }
G^^	-4 2 2	{ empilha valor de G }
^^	-4 2 2	{ calcula valor da subexpressão }
^	-4 4	{ calcula valor da subexpressão }
	256	