Universidade Federal do Rio Grande do Norte
Instituto Metrópole Digital

Estruturas de Dados Básicas I • IMD0029
◁ Projeto de Programação ▷

Programas BARES (Basic ARithmetic Expression Evaluator based on Stacks)

24 de setembro de 2015

Apresentação

O objetivo deste exercício de programação é motivar o uso das estrutura de dados estudada, *pilha* e *fila*, no contexto de uma aplicação real. A aplicação a ser desenvolvida é um avaliador de expressões aritméticas simples.

Sumário

1 Expressões

O programa **BARES** (*Basic ARithmetic Expression Evaluator based on Stacks*) deverá ser capaz de receber expressões aritméticas simples, formadas por:-

- *constantes numéricas inteiras* (-32.767 a 32.767);
- *operadores* ($+, -, /, *, \hat{}, \%$), com precedência descrita em Tabela 1; e
- parênteses.

Precedência	*Operadores*	*Associação*	*Descrição*
1	`()`	$\longrightarrow$	Quebra de precedência
2	`-` (unário)	$\longleftarrow$	Negação da constante inteira
3	`^`	$\longleftarrow$	Potenciação ou Exponenciação
4	`* / %`	$\longrightarrow$	Multiplicação, divisão, resto divisão inteira
5	`+ -`	$\longrightarrow$	Adição, subtração

Tabela 1: Precedência e ordem de associação de operadores em expressões *bares*.

Segue abaixo exemplos de expressões válidas para o *bares*:-

- `35 - 3 * (-2 + 5)^2`
- `54 / 3 ^ (12%5) * 2`
- `((2-3)*10 - (2^3*5))`
- `---3 + 4`

O fim de linha ('\n') será o indicador de fim de expressão, ou seja, o programa deverá receber uma expressão por linha de entrada de dados. Note que espaços em branco (código ascii 32) e tabulações (código ascii 9) podem aparecer antes da expressão, entre os termos da expressão ou após a expressão e devem ser ignorados pelo programa.

2 A Tarefa

Sua tarefa consiste em elaborar um programa em C++ denominado `bares.cpp` que deverá receber via arquivo uma ou mais expressões, uma por linha. O programa deverá, então, avaliar cada expressão e imprimir seu respectivo resultado na saída padrão, `std::cout`, ou em um arquivo texto de resultados informado pelo usuário. A forma geral de chamada o programa seria:-

```
$.\bares arquivo_entrada [arquivo_saida]
```

Por exemplo, a resposta que o programa deveria oferecer para as expressões exemplo da Seção 1 seria: 8, 12 e -50, e 1 (uma resposta por linha). Os detalhes sobre tratamento de erros e formatos de saída do programa estão descritos na Seção 4.

3 Solucionando o Problema

De maneira básica, o problema que este trabalho tenta resolver é o seguinte. Considerando a expressão $2 + 3 \times 6$, qual o seu resultado?

$$\underbrace{2+3}_{5}\times 6 \quad ou \quad 2+\underbrace{3\times 6}_{18}$$
$$\underbrace{\qquad\quad}_{30} \qquad\qquad \underbrace{\qquad\quad}_{20}$$

Sabemos que o correto é 20, visto que a multiplicação tem prioridade maior do que a adição. Por isso executamos a multiplicação primeiro e depois a adição. Porém, precisamos sistematicamente resolver esta ambiguidade por meio de um algoritmo. Como conseguir isso?

A estratégia de solução recomendada para este trabalho envolve mudança automatizada na forma de representar uma expressão aritmética. Esta mudança, descrita nas próximas seções, facilita o processamento (avaliação) de uma expressão na medida que elimina a ambiguidade inerente à representação infixa.

3.1 Notações de Expressões

Uma expressão para somar A e B pode ser descrita como $A + B$. Essa representação é denominada de forma **infixa**. Além dessa existem outras duas representações, a saber:-

$+\,A\,B$ **prefixa**

$A\,B\,+$ **posfixa**

Na notação prefixa o operador (no caso o '+') é introduzido antes de seus dois operandos (A e B). Já na notação posfixa[1] o operador aparece logo após seus dois operandos.

As regras que devem ser consideradas durante o processo de conversão são: i) as operações com a precedência mais alta são convertidas em primeiro lugar (quando existe ambiguidade) e; ii) uma expressão convertida para a forma posfixa deve ser tratada como um único operando. Veja na Tabela 2 alguns exemplos adicionais de conversão da forma infixa para a posfixa.

Forma Infixa	*Forma Posfixa*
$A + B$	$AB+$
$A + B - C$	$AB + C-$
$(A + B) * (C - D)$	$AB + CD - *$
$A\hat{}B * C - D + E/F/(G + H)$	$AB\hat{}C * D - EF/GH + /+$
$((A + B) * C - (D - E))\ \hat{}(F + G)$	$AB + C * DE - -FG+\hat{}$
$A - B/(C * D\ \hat{}E)$	$ABCDE\ \hat{}*/-$

Tabela 2: Exemplos de equivalências entre forma infixa e posfixa de expressões.

[1] Também conhecido como Notação Polonesa Reversa.

3.2 Separando os Termos de uma Expressão

Uma tarefa que antecede a conversão entre representações de expressões aritméticas é separação dos termos de uma expressão, denominada de *tokenização*.

Assumindo que uma expressão de entrada é fornecida por meio de uma cadeia de caracteres (por exemplo, `std::string`), a tokenização consiste em percorrer esta cadeia, caractere por caractere, separando seu termos em uma lista de *tokens*. Neste processo, caracteres em branco (código ascii 32) ou tabulações (código ascii 9) são ignorados e o fim de linha ('\n') indica o fim da expressão.

No contexto deste trabalho, um token pode ser um *operando* (constantes inteiras) ou um *operador* (uma das operações aritméticas válidas para o trabalho). Por exemplo, se a entrada for a expressão "$(2+3)*10/-5$", a tokenização deve produzir uma lista sequencial de tokens da seguinte forma:

$$\{\text{“(”, “2”, “+”, “3”, “)”, “*”, “10”, “/”, “-”, “5” }\}.$$

Muitas vezes o processo de tokenização é feito em conjunto com o processo de *parsing*. *Parsing* consiste em fazer uma *análise sintática* da expressão composta por tokens para saber se ela obedece a sintaxe que define uma expressão válida.

Durante o *parsing* vários error de formação de expressão devem ser detectados e sinalizados para o usuário. Existem diversas técnicas para realizar *parsing* de uma maneira eficiente. Porém, neste trabalho, sugere-se que seja adotada uma forma mais simples, visto que as expressões são formadas apenas por constantes inteiras e alguns caracteres representando as operações aritméticas suportadas. Como você abordaria este problema com os conhecimento que possui hoje, ou seja, sem utilizar técnicas clássicas de *parsing*?

3.3 Transformações Entre Notações

Como mencionado anteriormente, a estratégia de solução sugerida envolve a transformação da expressão original do formato *infixo*, o qual é mais natural para o usuário fornecer uma entrada, para o formato *posfixo*, o qual é mais "natural" para um processamento sem ambiguidades. A transformação de um formato para outro apresenta três vantagens:-

1. Durante o procedimento de transformação é possível detectar alguns erros de formação de expressão;
2. A expressão no formato posfixo não necessita da presença de delimitadores (parênteses) por ser uma representação *não-ambígua*;
3. O algoritmo para avaliar uma expressão posfixa é mais simples do que um algoritmo para avaliar uma expressão infixa.

3.4 Convertendo Expressões: de infixa para posfixa

Para realizar a avaliação da expressão (descrito na Seção 3.5), faz-se necessário, primeiramente, converter a expressão de infixa para posfixa. Essa conversão deve ser feita de tal forma a

lidar de forma correta com, digamos, os casos "$A + B * C$" e "$(A + B) * C$"—produzindo, respectivamente, as expressões "$ABC * +$" e "$AB + C*$".

Ao analisar os casos acima percebe-se que o algoritmo de conversão deve possuir algum tipo de mecanismo para armazenar os operadores temporariamente de tal forma que a regra de precedência de operadores seja respeitada. Esse mecanismo de armazenamento também será uma estrutura de dados do tipo *pilha*.

O Algoritmo 1 apresenta uma forma de converter uma expressão (sem parênteses[2]) no formato infixo para o posfixo. Para tanto precisamos de uma função denominada `prcd(op1,op2)` — onde `op1` e `op2` são operadores — que retorna `true` se `op1` tiver precedência sobre `op2` ou ambos tiverem a mesma precedência, e `false` caso contrário. Por exemplo, `prcd('*','+')` e `prcd('-','+')` retornam `true`, enquanto `prcd('+','*')` é `false`.

Algoritmo 1 Conversão de expressão no formato infixo para posfixo.

Entrada: Fila de '`Símbolo`' representando uma expressão no formato infixo.
Saída: Fila de '`Símbolo`' representando uma expressão no formato posfixa equivalente.

```
1:  função Infx2Posfx(fila no formato infixo): fila no formato posfixo
2:      enquanto não chegar ao fim da fila de entrada faça
3:          remover símbolo da fila de entrada em symb
4:          se symb for operando então
5:              enviar symb diretamente para fila de saída
6:          senão
7:              enquanto Pilha não estiver vazia e símbolo do topo (topSymb) ≥ symb faça
8:                  se topSym ≥ symb então
9:                      remover topSym e enviar para fila de saída
10:             Empilhar symb     # depois que retirar operadores de precedência ≥, inserir symb
        # descarregar operadores remanescentes da pilha
11:     enquanto Pilha não estiver vazia faça
12:         remover símbolo da pilha e enviar para fila de saída
13:     retorna fila de saída na forma posfixa
```

Um dos desafios é pensar quais modificações seriam necessárias para que o Algoritmo 1 possa acomodar o uso do '`-`' unário (como em "$-2 + 5$") e de parênteses.

3.5 Avaliando Uma Expressão Posfixa

Para realizar a avaliação de uma expressão na forma posfixa pode-se utilizar uma estrutura de dados do tipo *pilha*. Cada vez que um **operando** (i.e. uma constante inteira) é encontrado na expressão o mesmo deve ser introduzido na pilha. Quando um **operador** (i.e. $+$, $\hat{}$, $*$, etc.) é encontrado na expressão, os dois elementos no topo da pilha são seus operandos. Portanto,

[2]Você deve adaptar o algoritmo para tratar o uso de parênteses.

devemos retirar esses dois elementos da pilha, realizar a operação indicada pelo operador[3] e, a seguir, (re)introduzir o resultado de volta na pilha, tornando-o disponível para uso como operando do próximo operador. Confira o Algoritmo 2.

Um dos desafios é pensar que modificação é necessária para que o Algoritmo 2 possa acomodar o operador '-' unário, visto que ele precisa de apenas 1 operando e não 2 como as demais operações.

Algoritmo 2 Avaliação de expressão no formato posfixo.

Entrada: Fila de 'Símbolo' representando uma expressão no formato posfixo.
Saída: Resultado da expressão avaliada.

```
1:  função AvalPosfixa(FPosfixa: FilaDeSímbolo): inteiro
2:      var symb: Símbolo                          # símbolo atual a ser analisado
3:      var OPn: PilhaDeInteiro                    # pilha de operandos
4:      var opnd1, opnd2: Operando                 # operandos auxiliares
5:      var resultado: inteiro                     # recebe o resultado de operação
6:      enquanto não FPosfixa.isEmpty() faça       # não chegar ao fim da fila...
7:          FPosfixa.dequeue(symb)
8:          se symb.ehOperando() então             # é operando?
9:              OPn.push(symb.getValue)            # empilha operandos
10:         senão
11:             OPn.pop(opnd2)                     # (inverter) recupera 2º operando
12:             OPn.pop(opnd1)                     # (inverter) recupera 1º operando
13:             resultado ← Aplicar symb à opnd1 e opnd2   # um ''caso'' para cada operador
14:             OPn.push(resultado)    # armazenar resultado; fila pode estar em processamento
15:     OPn.pop(resultado)                         # recuperar o valor final da pilha e...
16:     retorna resultado                          # ...retorna o valor inteiro do símbolo
```

4 Tratamento de Erros

Note que em caso de haver algum problema com a expressão (e.g. escopo aberto, operador inválido, falta de operando, etc.) o programa deverá indicar qual o erro ocorrido e em que posição (coluna) da expressão.

De uma forma geral é possível separar os erros em categorias: entrada de dados inválidas (e.g. números em ponto flutuante ou fora da faixa dos inteiros), elaboração incorreta da expressão (e.g. falta de parênteses e símbolos não reconhecido), e erro na avaliação da expressão (e.g. divisão por zero, falta operando).

O conjunto **mínimo** de erros que devem ser tratados são os seguintes[4]:

1. **Constante numérica inválida**: Um dos operandos da expressão está fora da faixa

[3] Cuidado com a ordem dos operandos, pois a pilha tem comportamento *LIFO* (*Last In, First Out*).
[4] O símbolo '␣' é utilizado para representar espaço em branco, não aparecendo de fato na expressão.

permitida.
Ex.: $1000000 - 2$, coluna 1.

2. **Falta operando**: Em alguma parte da expressão está faltando um operando.
   Ex.: $2+$, coluna 2; ou $2 ** 3$, coluna 3.
3. **Operando inválido**: Existe um operando que não é uma constante numérica válida.
   Ex.: $3 * d$, coluna 5.
4. **Operador inválido**: Existe um símbolo correspondente a um operador que não está na lista de operadores válidos.
   Ex.: $2 = 3$, coluna 3; ou $2.3 + 4$, coluna 2.
5. **Falta operador**: Aparentemente o programa encontrou um operando válido mas "perdido" (isto é, sem um operador associado) na expressão.
   Ex.: $2 * 3 4$, coluna 5.
6. **Fechamento de escopo inválido**: Existem um parêntese fechando sem ter um parêntese abrindo correspondente.
   Ex.: $)2 - 4$, coluna 1; ou $2 - 4)$, coluna 6.
7. **Escopo aberto**: Existe um parêntese de abertura '(' sem um parêntese de fechamento ')' correspondente.
   Ex.: $((2\%3) * 8$, coluna 1.
8. **Divisão por zero**: Houve divisão cujo quociente é zero.
   Ex.: $3/(1 - 1)$, coluna 4 (os parênteses são ignorados).

Na saída do programa—seja em arquivo ascii ou na saída padrão `std::cout`—a mensagem de erro para o usuário deve ser a mais precisa e objetiva possível, permitindo que o usuário identifique a fonte de erro prontamente. Por isso é importante informar a coluna de ocorrência do erro. Cada dupla pode elaborar a mensagem e formato de exibição que achar mais apropriado.

Em linhas gerais para cada expressão de entrada (1 por linha) é possível gerar no arquivo de saída ou na saída padrão:-

- Uma linha com *apenas* com o resultado da expressão, se a expressão estiver sintaticamente correta; ou
- Uma ou mais linhas, cada uma contendo um único erro de sintaxe detectado com sua respectiva coluna da ocorrência.

5 Avaliação do Programa

Para a implementação deste projeto **é obrigatório** a utilização das classes pilha, fila e lista sequencial que foram apresentadas em sala de aula. Não serão aceitas soluções que utilizem

as estruturas de dados da biblioteca externas, como STL (e.g. `list`, `stack`, `vector`, etc.) ou `boost`, por exemplo.

O programa completo deverá ser entregue sem erros de compilação, testado e totalmente documentado. O projeto será avaliado sob os seguintes critérios:-

- Trata corretamente os argumentos de linha de comando (5%);
- Lê expressões de um arquivo ascii e cria corretamente uma lista de tokens (10%);
- Converte corretamente expressões do formato infixo para posfixo (20%);
- Trata corretamente o uso de parênteses e '`-`' unário (5%);
- Avalia corretamente expressão no formato posfixo (15%);
- Detecta corretamente o conjunto mínimo de erros solicitados (15%);
- Gera a saída conforma solicitado (15%); e
- Código é organizado em classes (15%).

A pontuação acima não é definitiva e imutável. Ela serve apenas como um guia de como o trabalho será avaliado em linhas gerais. É possível a realização de ajustes nas pontuações indicadas visando adequar a pontuação ao nível de dificuldade dos itens solicitados.

Os itens abaixo correspondem à descontos, ou seja, pontos que podem ser retirados da pontuação total obtida com os itens anteriores:-

- Presença de erros de compilação e/ou execução (até -20%)
- Falta de documentação do programa com Doxygen (até -10%)
- Vazamento de memória (até -10%)
- Falta de um arquivo texto `README` contendo, entre outras coisas, identificação da dupla de desenvolvedores; instruções de como compilar e executar o programa; lista dos erros que o programa trata; e limitações e/ou problemas que o programa possui/apresenta, se for o caso (até -20%).

Boas práticas de programação

Recomenda-se fortemente o uso das seguintes ferramentas:-

- Doxygen: para a documentação de código e das classes;
- Git: para o controle de versões e desenvolvimento colaborativo;
- Valgrind: para verificação de vazamento de memória;
- gdb: para depuração do código; e
- Makefile: para gerenciar o processo de compilação do projeto.

Recomenda-se também que sejam realizados testes unitários nas suas classes de maneira a garantir que elas foram implementadas corretamente. Procure organizar seu código em várias pastas, conforme vários exemplos apresentados em sala de aula, com pastas como `src` (arquivos `.cpp`), `include` (arquivos `.h`), `bin` (arquivos `.o` e executável) e `data` (arquivos de entrada e saída de dados).

Uma forma de validar o seu programa é inserir diretivas de compilação condicional para compilar o seu projeto ora usando suas classes (pilha, fila, lista, etc.), ora usando classes equivalentes do STL (`stack` , `vector` , `list` , etc.). Esta estratégia permite isolar erros no programa BARES de erros na implementação das classes básicas.

6 Autoria e Política de Colaboração

O trabalho pode ser realizado **individualmente** ou em **duplas**, sendo que no último caso é importante, dentro do possível, dividir as tarefas igualmente entre os componentes.

Qualquer equipe pode ser convocada para uma entrevista. O objetivo da entrevista é duplo: confirmar a autoria do trabalho e determinar a contribuição real de cada componente em relação ao trabalho. Durante a entrevista os membros da equipe devem ser capazes de explicar, com desenvoltura, qualquer trecho do trabalho, mesmo que o código tenha sido desenvolvido pelo outro membro da equipe. Portanto, é possível que, após a entrevista, ocorra redução da nota geral do trabalho ou ajustes nas notas individuais, de maneira a refletir a verdadeira contribuição de cada membro, conforme determinado na entrevista.

O trabalho em cooperação entre alunos da turma é estimulado. É aceitável a discussão de ideias e estratégias. Note, contudo, que esta interação **não** deve ser entendida como permissão para utilização de código ou parte de código de outras equipes, o que pode caracterizar a situação de plágio. Em resumo, tenha o cuidado de escrever seus próprios programas.

Trabalhos plagiados receberão nota **zero** automaticamente, independente de quem seja o verdadeiro autor dos trabalhos infratores. Fazer uso de qualquer assistência sem reconhecer os créditos apropriados é considerado **plagiarismo**. Quando submeter seu trabalho, forneça a citação e reconhecimentos necessários. Isso pode ser feito pontualmente nos comentários no início do código, ou, de maneira mais abrangente, no arquivo texto `README`. Além disso, no caso de receber assistência, certifique-se de que ela lhe é dada de maneira genérica, ou seja, de forma que não envolva alguém tendo que escrever código por você.

7 Entrega

Você deve submeter um único arquivo com a compactação da pasta do seu projeto. Se for o caso, forneça também o link Git para o seu projeto. O arquivo compactado deve ser enviado **apenas** através da opção Tarefas da turma Virtual do Sigaa, em data divulgada no sistema.

◀ FIM ▶