Calculadora de Expressões
Trabalho 1 - Estruturas de Dados

André Cássio Barros de Souza, 16/0111943
Giovanni M Guidini, 16/0122660

[1]Dep. Ciência da Computação – Universidade de Brasília (UnB)
CiC 116319 - Estruturas de Dados - Turma A

andreloff15@hotmail.com, giovanni.guidini@gmail.com

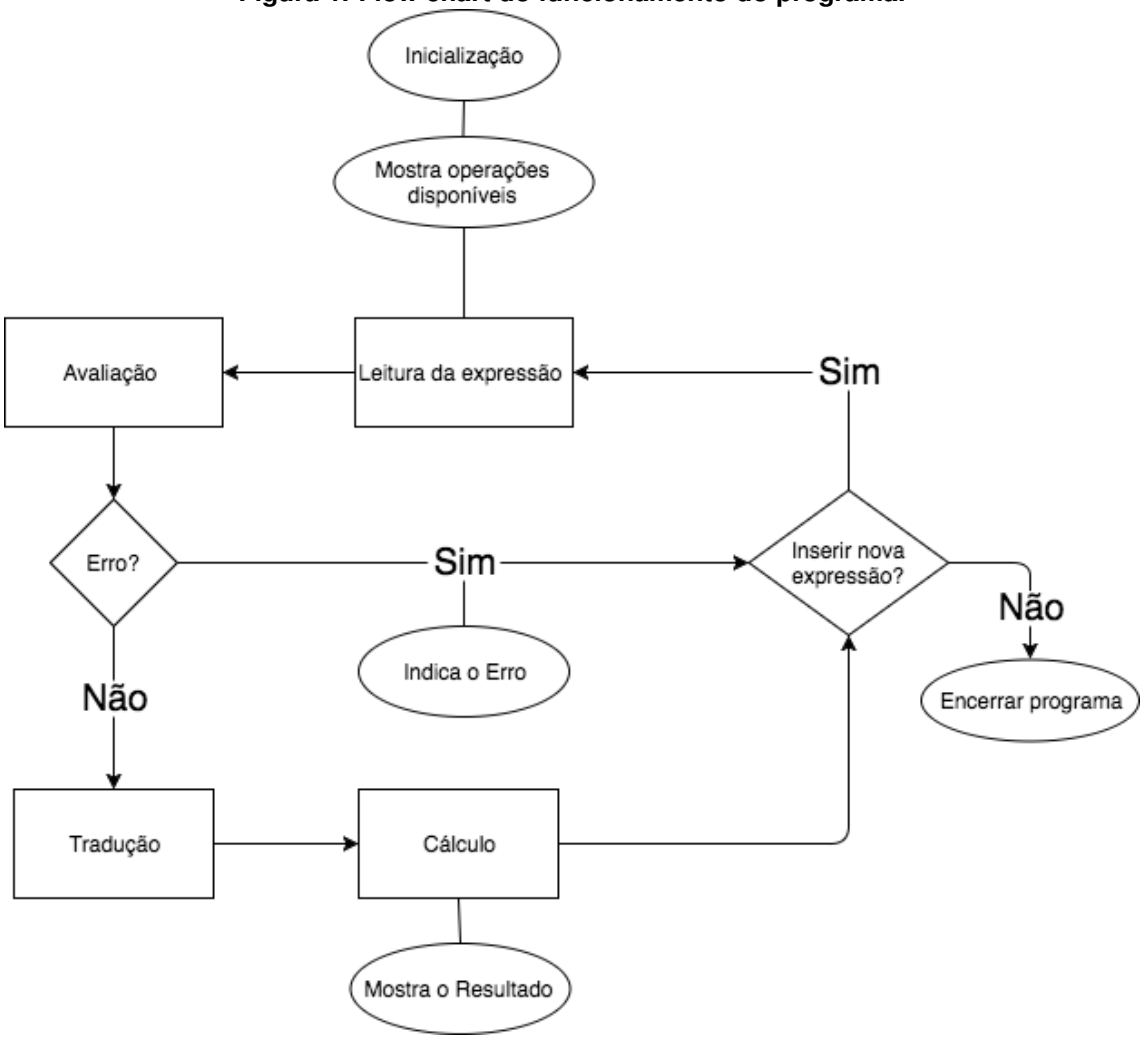
Abstract. *The expression calculator is an algorithm that implements a simple calculator for mathematic expressions containing only numbers, received in infix order. It performs the four basic arithmetic operations - addition, subtraction, multiplication and division - and also accepts the use of parenthesis. It also transforms the infix expression to its postfix translation.*

Resumo. *A calculadora de expressões é um algoritmo que implementa uma calculadora simples para avaliação de expressões matemáticas contendo números apenas, recebida na forma infixa. Ela faz as quatro operações aritméticas simples - adição, subtração, multiplicação e divisão - e também aceita o uso de parêntesis. Também faz a tradução da expressão para a forma pós-fixa.*

1. Introdução

A calculadora de expressões funciona em três partes. A saber: leitura da entrada e sua validação, tradução da entrada infixa para sua forma pós-fixa, e finalmente a avaliação da expressão e computação do resultado. Cada uma destas partes foi implementada usando estruturas de dados apropriadas para criar um algoritmo rápido, eficiente, robusto e, principalmente, correto. Além disso o programa permite que várias expressões sejam inseridas por execução, isto é, ele não termina ao fim do cálculo das expressões, e sim quando o usuário indicar que não quer mais inserir expressões. O funcionamento do programa é representado pela figura 1.

Figura 1. Flow chart do funcionamento do programa.



Outro aspecto importante são as Estruturas de Dados utilizadas. Foram utilizadas principalmente listas encadeadas e stacks. As operações relacionadas a estas estruturas estão definidas nas bibliotecas locais utilizadas (linkedlist.h e stack,h) no código. Cópias do código destas bibliotecas podem ser encontradas no apêndice A.

Avaliar uma expressão contendo várias operações diferentes, mesmo que sejam expressões elementares e aparentemente simples não é tão fácil quanto parece. De fato, o computador não consegue fazer a leitura das expressões da mesma forma que nós fazemos. Além disso as operações só podem ser feitas uma por vez, logo, a ordem correta precisa ser estabelecida de antemão. Em suma, calcular a expressão é o último de uma série de passos que precisam ser realizados cuidadosamente para evitar erros. A seguir uma descrição geral dos passos que o programa executa para validar e calcular expressões.

1.1. Passo 1 - Validação

O primeiro passo, então, é validar a expressão recebida sintaticamente. Nosso código faz isso avaliando 3 aspectos principais neste ponto do código:

1. **Avaliação de escopos** - verifica se todo parêntesis aberto também é corretamente

fechado. Caso existam problemas o programa retorna erro e não computa a expressão.

2. **Existência de números** - verifica se existe pelo menos um número na expressão. Se não houver nenhum número o programa retorna um erro e não computa a expressão.
3. **Existência de caracteres inválidos** - verifica a existência de letras. A calculadora não suporta inserção de variáveis, somente números e operações. Se alguma letra ou outro caractere inválido (i.e. , %, $, #, etc) é detectado o programa retorna erro e não computa a expressão. Os caracteres válidos são {+, -, *, /}.

Se todos estes passos forem concluídos e a expressão for sintaticamente aceitável passamos para o segundo passo, que é a transformação da expressão de sua forma infixa (como é recebida) para a forma pós-fixa (como o computador tem mais facilidade de decifrá-la).

1.2. Passo 2 - Tradução

É importante saber que a maior diferença entre estas duas formas de se escrever uma expressão matemática é que a forma pós-fixa não utiliza parêntesis, as operações e seus operandos já são escritos na sequência em que devem ser avaliados. Isso não acontece na forma infixa, onde cada operação tem um grau maior ou menor de prioridade, e ainda existem os parêntesis que mudam estas prioridades. Para exemplos de expressões infixas e pós-fixas, e links com mais informações sobre o assunto, ver apêndice B.

Para implementar esta parte do trabalho seguimos o algoritmo descrito em [1]. Como uma descrição mais detalhada será fornecida mais a frente neste relatório, por hora basta saber que a tradução da expressão infixa para pós-fixa requer a utilização de uma pilha auxiliar e a criação de uma nova lista. Ao final do processo a antiga lista é liberada para evitar vazamentos na memória e economizar espaço que estaria sendo desperdiçado. É nesta parte do programa que a prioridade das operações e a existência ou não de parêntesis é avaliada, e todas as operações são reorganizadas de acordo com esta prioridade.

1.3. Passo 3 - Avaliação

O passo final é calcular o valor da expressão, usando para isso a nova expressão gerada no passo 2. Para tanto também seguimos, de forma geral, o algoritmo descrito em [1]. Este algoritmo faz uso de uma pilha auxiliar para armazenar os operandos e resultados de cálculos executados até que todas as operações da expressão sejam computadas. Algumas particularidades das operações tiveram de ser consideradas. Estas estão melhor detalhadas na seção 2.5.

2. Implementação

A implementação do trabalho como um todo visou facilitar o entendimento do código e sua futura manutenção, sempre prezando por eficiência e rapidez. Neste sentido, muitas funções foram criadas para encapsular o código em várias partes diferentes. Elas estão listadas[1] na seção 2.1. Alguns problemas inerentes ao problema a ser resolvido foram

[1] Somente as funções do arquivo principal, `calculadora.h`, estão listadas nesta seção. As funções usadas nas bibliotecas locais podem ser encontradas no apêndice A.

solucionados através de convenções simples presentes na implementação. Estes serão comentados ao longo das seções 2.2, 2.3, 2.4 e 2.5.

2.1. Lista de Funções

Funções que cuidam da entrada e saída de dados e interação com o usuário:

- `showHelp()` - função void que apresenta as operações disponíveis ao usuário ao abrir o programa. Existe para diminuir erros causados por usuários.
- `getB()` - função que retorna um int b de controle. para a continuação ou encerramento do programa após a conclusão do processamento de uma expressão.
- `readInput()` - função faz a leitura da expressão e a armazena, carctere por caractere, em uma lista simplesmente encadeada. Retorna um ponteiro para o início da lista criada.

Função para Validação:

- `errorCheck(t_lista* l)` - função recebe a expressão lida através da lista l, e a percorre para fazer as verificações de escopo, existência de números e ausência de caracteres inválidos. Retorna 0 se expressão estiver válida e 1, 2, ou 3 dependendo do erro encontrado.

Função para tradução:

- `inToPos(t_lista* l)` - função recebe a expressão válida através da lista l. Em seguida percorre a lista toda executando a transformação para pós-fixa, utilizando uma pilha auxiliar. Retorna uma nova lista com a expressão pós-fixa. Limpa lista anterior para evitar vazamentos de memória.

Funções para Avaliação:

- `int strLen(char* ch)` - função retorna o tamanho real do número a ser calculado. Existe um offset devido ao expaço ou arroba no final do número para indicar seu sinal.
- `long long charToInt (char* ch)` - função recebe um string contendo o número e retorna o valor do número. Caso o número seja negativo, o valor retornado também é negativo.
- `int Operacao(t_numStack* jooj, char op, char field)` - função que realiza as diferentes operações da expressão. Recebe a stack numérica, a operação a ser realizada, e se ela será uma operação entre inteiros ou frações. Retorna 0 se a operação for bem sucedida ou 1 se não for.
- `void Resolve(t_lista* l)` - função que separa os números da expressão e indica qual a próxima operação a ser realizada. No fim de todas as operações mostra o resultado na tela.

2.2. Entrada e Saída de Dados

A entrada e saída de dados neste programa é relativamente simples. Consiste basicamente na leitura da expressão, apresentação do resultado, e pedir input do usuário e emitir warnings. Como as últimas destas operações são relativamente simples, vamos enfatizar a leitura da expressão a ser calculada. Essa leitura é feita pela função `readInput()`. A implementação da função é mostrada no código 1.

Foi utilizada uma lista de caracteres para garantir flexibilidade e aceitar expressões de qualquer tamanho sem alocar espaço desnecessário de memória. Repare também que as operações das linhas 11 e 16 se tornam extremamente simples nesta implementação, e cobre um erro bastante comum na inserção de expressões matemáticas no computador devido à naturalidade de se omitir aquele sinal de multiplicação. Corrigindo este pequeno erro menos expressões são rejeitadas.

Repare também que a linha 7 ”come”os espaços. Isso será importante mais à frente na hora de separar os números - uma vez que números de mais de um algarismo ficam armazenados caractere por caractere - na expressão pós-fixa.

```
1  t_lista* readInput(){
2    t_lista* l = newList();
3    char ch;
4    printf("Please, insert expression.\n");
5
6    while(scanf("%c", &ch), ch != '\n'){
7      if(ch != ' '){
8        if(!isListEmpty(l)){  // evitar seg fault
9          if(ch == '('){  // adiciona sinal de * caso a expressao seja,
10           if(isdigit(l->fim->data)){  //  por exemplo, a(b+c).
11             insertFim(l, '*');
12           }
13         }
14         else if(isdigit(ch)){// adiciona sinal de * caso a expressao
15           if(l->fim->data == ')'){  //  seja do tipo (b+c)a.
16             insertFim(l, '*');
17           }
18         }
19       }
20       insertFim(l, ch);
21     }
22   }
23   return l;
24 }
```

Listing 1. Implementação da função readInput()

2.3. Validação da Expressão

Para a validação da expressão foi utilizada a função `errorCheck()`, cuja implementação pode ser vista no código 2. Esta função recebe uma lista encadeada contendo a mensagem a ser validada e retorna um inteiro $i \in [0, 3]$ que representa um erro ou não.

O erro 1 indica problemas no escopo, sendo detectado através de uma stack que armazena parêntesis de abertura e desempilha toda vez que achar um parêntesis de fechamento. Se houver uma tentativa de desempilhar a stack vazia, ou se ela não estiver vazia ao final da varredura, então o escopo está errado. Ele pode indicar também erro semântico, no caso de haverem dois operadores em sequencia. Exemplos de expressões erradas que seriam detectadas por este erro seriam $(1 - 2) * (3 + 5$, erro de escopo, e $1 - 3 + +4$, dois operadores seguidos.

O erro 2 indica que a expressão não apresenta nenhum número. Não havendo valor a ser calculado, um erro é emitido. Exemplo: $()()()(())$.

O erro 3 indica que algum caractere inválido foi inserido, como uma letra, por exemplo: $a - b + c$. Repare que o teste realizado na linha 9 usa o campo prioridade. Números possuem prioridade 1000, operadores e parêntesis possuem prioridade 0, 1 ou 2 dependendo do caractere, outros caracteres possuem prioridade 5. As prioridades estão definidas em `stack.h`.

Note que a função meramente retorna o número do erro. A mensagem associada ao erro é emitida na função `main()`.

```
1  int errorCheck(t_lista* l){
2    t_elemento* it = l->inicio;
3    t_stack* err = newStack();
4    int b = 0;  // flag de numeros
5    while(it != NULL){
6      if(isdigit(it->data)){
7        b = 1;
8      }
9      else if(it->prioridade == 5){ // flag para caracteres invalidos
10       return 3;
11     }
12     else if((it->prioridade == 1) || (it->prioridade == 2)){
13       if((it->prox->prioridade == 1) || (it->prox->prioridade == 2))
14       // operadores seguidos
15       return 1;
16     }
17     else if(it->data == '('){
18       push(err, it->data);
19     }
20     else if(it->data == ')'){
21       if(!isStackEmpty(err)){
22         pop(err);
23       }
24       else{
25         // tentativa de remover de stack vazia significa erro.
26         return 1;
27       }
28     }
29     it = it->prox;
30   }
31   if(b == 0){   // significa que nao ha nenhum numero na expressao
32     return 2;
33   }
34   if(!isStackEmpty(err))  // se a stack nao estiver vazia neste ponto,
       significa erro.
35   return 1;
36
37   else          // escopos estao ok
38   return 0;
39 }
```

Listing 2. Implementação da função errorCheck()

2.4. Tradução

A tradução é realizada pela função `inToPos()`. Esta função recebe uma lista simplesmetne encadeada contendo a mensagem a ser traduzida e retorna outra lista com a

mensagem traduzida. A lista retornada não é a mesma que a lista recebida, sendo que esta última é destruída. Não será mostrada a implementação da função neste relatório porque ela é demasiado extensa, mas ela pode ser encontrada no código completo (ver apêndice C).

Para realizar a tradução uma nova lista é criada, bem como uma stack auxiliar e um ponteiro para t_elemento (os elementos da lista). O ponteiro começa no início da expressão e vai até o final, sendo que a cada iteração uma ação é tomada dependendo do caractere em questão.

Números são passados diretamente para a nova lista na ordem em que aparecem. Note, no entanto, que números com mais de um algarismo estão armazenados em mais de um elemento da lista. Como são adicionados na nova lista na ordem em que aparecem a sequência dos algarismos não é alterada, mas não poderíamos saber quando um número acaba e o próximo começa. Para resolver este problema foi adicionada uma flag b, inicializada com o valor 1. Quando um algarismo é adicionado a flag recebe o valor 1. Quando o caractere analisada é um operador a flag recebe o valor 0. Caso um novo número deva ser adicionado e $b = 0$ no momento de adicioná-lo, um espaço é inserido antes do número, marcando o fim do número adicionado anteriormente. Isso funciona pois na notação infixa os operadores sempre vêm entre os operandos.

Operadores binários não são adicionados imediatamente na expressão de saída. Eles são copiados para a stack auxiliar, pois a ordem correta de operações deve ser estabelecida. Para isso, toda vez que um novo operador está para ser adicionado na stack verificamos se no topo dela existe um operador de prioridade maior. Caso haja, ele deve ser removido da stack e adicionado na nova lista, pois aquela operação deve ser realiada antes da nova operação que agora está no topo da stack. Cada operador é empilhado e desempilhado exatamente uma vez.

Os operadores unários deram um pouco mais de trabalho. Em primeiro lugar precisamos diferenciar os operadores + e - binários dos unários. Isso é feito avaliando-se o lugar em que eles estão inseridos, de tal forma que os operadores unários satisfazem alguma destas condições:

- São precedidos por parêntesis de abertura;
- São precedidos por outro operador;
- São o primeiro caractere da expressão de entrada.

Satisfeita alguma destas condições, caso o operador fosse - era adicionado um caractere diferente (usamos @) na pilha de operadores. A prioridade deste operador é maior do que de qualquer outro operador, pois ele deve vir imediatamente após o próximo número que será adicionado à expressão de saída. O operador unário + era simplesmente ignorado, pois não afeta a expressão.

Parêntesis alteram a prioridade de operações. Por isso no algoritmo eles criam uma espécie de sub-stack dentro da stack auxiliar da seguinte maneira: quando um parêntese de abertura é encontrado ele é adicionado no topo da stack auxiliar. Sabemos que as operações que vierem dentro deste parêntese serão realizadas antes das operações que por acaso estão na stack abaixo dele. Como estas operações serão as próximas inseridas podemos considerar o parêntese de abertura como o fim da sub-stack, que acaba com o parêntese de fechamento. Ao acharmos o fechamento removemos os elementos sobre

o parêntese de abertura até que ele seja removido da stack auxiliar. Note que se vários parênteses estiverem aninhados o algoritmo ainda funciona, seria apenas o caso de criar sub-stacks nas sub-stacks da auxiliar[2].

Apesar de os parêntesis terem prioridade maior do que outros operadores, para o algoritmo sua prioridade é menor. Isso foi feito para evitar que os operadores que vêm entre os parêntesis removessem o parêntese de abertura erroneamente da stack quando fossem inseridos, perdendo o escopo em que deveriam estar inseridos.

Ao fim da função, logo antes de retornar `t_lista* nova` é chamada a função `clearList(l)`, sendo l a lista que contém a mensagem original. Isto destrói a lista l, que já não será mais utilizada, evitando memory leaks no programa.

2.5. Avaliação

Esta parte do código funciona em torno de duas funções principais, `Resolve()` e `Operacao()`, e mais as outras funções auxiliares listadas na seção 2.1. As funções desta parte do código também são bastante extensas, então sua implementação será somente discutida aqui. Recomendamos ver o apêndice C para ver a implementação delas.

Esta foi, sem dúvida, a parte mais complicada do código. Ela requeriu que um novo tipo de stack fosse criado, a stack numérica, onde os números da expressão traduzida, que estão armazenados na forma de caracteres são adicionados como números. Os elementos desta stack, no entanto, podem ser números reais também, depois de certas operações. Por isso eles possuem 3 campos para valores: `whole`, que armazena um valor inteiro, `frac`, que armazena um valor real, e `field`, que indica qual a informação relevante que o número carrega, sendo `'i'` para inteiro e `'d'` para real. OS valores numéricos são inicializados como 0 nos dois campos.

O desafio desta parte foi co-administrar uma lista de caracteres que continha números e operações com uma stack de números que poderiam ser de dois tipos diferentes e mudar de tipo ao longo das operações. No fim das contas, fizemos uma função em que cada número é encontrado e adicionado à stack numérica apenas uma vez, e função de execução de operações em tempo constante, sem perder os resultados parciais das operações.

Pois bem, a função que administra a avaliação da expressão é a função `Resolve()`. Esta função recebe a expressão traduzida e possui duas funcionalidades principais: reconhecer, separar e adicionar os números que estão como caracteres na expressão traduzida para a pilha de números; reconhecer as operações a serem realizadas e pedir sua execução.

Para isso um iterador percorre a expressão, armazenando os dígitos encontrados em um vetor. Ao encontrar um espaço ou arroba o programa sabe que o número acabou. Então o número é passado para as funções auxiliares e por fim adicionado na stack numérica. Ao encontrar um operador, os operandos daquela operação já estão na stack, então é só uma questão de pedir a execução. No fim, o último número na stack numérica é o resultado final.

[2]A analogia das sub-stacks foi feita para esclarecer o funcionamento do algoritmo usando a ideia de modularidade. Nenhuma nova stack é realmente criada no processo de tradução, mas pode-se abstrair que isso acontece por causa do funcionamento do algoritmo, como foi explicado.

Quando a operação é chamada, é passado o código da operação e um campo indicando se ela deverá ser feita entre inteiros ou reais. Os dois números são removidos da stack numérica, e decididos entre seus campos inteiros ou reais seguindo o campo field. Os valores são adicionados em novas variáveis auxiliares de acordo com a operação. Elas passam por inspeção de overflow.

Operações de soma, subtração e multiplicação trabalham com inteiros quando especificado field i, mas a divisão é sempre entre números reais. Se o valor dos operandos for inteiro antes da divisão, eles serão armazenados em variáveis do tipo double. Ela passa por inspeção para evitar divisão por zero. Quando especificado field d, todas as operações usam variáveis do tipo double.

Quando a operação é finalizada, o valor final é adicionado ao topo da stack numérica.

2.6. Análise de Complexidade

A complexidade de tempo deste algoritmo depende exclusivamente da entrada. A leitura, validação, tradução e cálculo fazem uma varredura na lista uma vez cada, logo o tempo de execução dessas funções é O(n). O tempo de execução do programa para uma expressão, desde a leitura até mostrar o resultado, é O(n). Não é possível fazer um algoritmo assintoticamente mais rápido do que este para este problema.

A complexidade de espaço depende da função sendo executada, mas não é maior do que o espaço de armazenamento de 2 listas encadeadas de tamanho n e uma stack também de tamanho n. Portanto o custo de espaço também é O(n).

2.7. Testes Realizados

Realizamos diversos testes com diferentes expressões para garantir que nossa calculadora fosse flexível e correta. Para cada teste é mostrado um print do input e output daquele teste. Testes semelhantes são mostrados em sequência. Testes são separados por objetivo, primeiro os testes de detecção de erro, depois os testes com operações com inteiros, depois os testes com operações com frações, seguidos por testes que misturam ambos e testes com sinais unários, para garantir a flexibilidade do algoritmo.

```
Teste #1
Please, insert expression.
1 + (2 * 3
expressao apresenta erros.

Teste #2
Please, insert expression.
1 + 2 * 3)
expressao apresenta erros.

Teste #3
Please, insert expression.
(((())))
expressao sem numeros.

Teste #4
Please, insert expression.
1 + 2*a + 3
```

```
expressao apresenta caractere invalido

Teste #5
Please, insert expression.
1+ (2/0)
expressao aceita.
---DIVISAO POR ZERO!---

Teste #6
Please, insert expression.
100000 * 100000
expressao aceita.
---OVERFLOW!---
```

Listing 3. Testes realizados para a detecção de erros

Nos testes #5 e #6 foi mostrado `expressao aceita` pois os primeiros erros são detectados apenas sintaticamente. Os dois últimos erros, devido à sua natureza semântica são detectados durante os cálculos da expressão.

```
Teste #7
Please, insert expression.
1 + 2(3 - 4) + 5
expressao aceita.
Resultado = 4

Teste #8
Please, insert expression.
1 - 2 + 4 - 6 + 7 + 8 - 10
expressao aceita.
Resultado = 2

Teste #9
Please, insert expression.
100 - 50 * (24 - 25)
expressao aceita.
Resultado = 150

Teste #10
Please, insert expression.
100000-99999
expressao aceita.
Resultado = 1
```

Listing 4. Testes realizados com inteiros

```
Teste #11
Please, insert expression.
(1/2) + (5/4)
expressao aceita.
Resultado = 1.750000

Teste #12
Please, insert expression.
(1/4)/(1/2)
expressao aceita.
Resultado = 0.500000
```

```
12
13 Teste #13
14 Please, insert expression.
15 (2/3) * (3/6) - (7/8) + (4/5)
16 expressao aceita.
17 Resultado = 0.258333
18
19 Teste #14
20 Please, insert expression.
21 26/13 + 10/5
22 expressao aceita.
23 Resultado = 4.000000
```

Listing 5. Testes realizados com frações

Repare que no teste #14, apesar da resposta ser inteira, o programa a trata como um valor real.

```
1 Teste #15
2 Please, insert expression.
3 1 - 3 * (5--4)/8
4 expressao aceita.
5 Resultado = -2.375000
6
7 Teste #16
8 Please, insert expression.
9 - - - - 2
10 expressao aceita.
11 Resultado = 2
12
13 Teste #17
14 Please, insert expression.
15 2*-3
16 expressao aceita.
17 Resultado = -6
18
19 Teste #18
20 Please, insert expression.
21 2 * (-3) + (1/2)
22 expressao aceita.
23 Resultado = -5.500000
24
25 Teste #19
26 Please, insert expression.
27 3 * --4 + (-(-(-8))) * 7/7
28 expressao aceita.
29 Resultado = 4.000000
30
31 Teste #20
32 Please, insert expression.
33 13000 - 400 + 45345 * -2
34 expressao aceita.
35 Resultado = -78090
```

Listing 6. Outros testes

3. Conclusão

A calculadora de expressões é um algoritmo rápido, eficiente e robusto que faz bom uso de Estruturas de Dados aprendidas ao longo das aulas teóricas de ED. Seguindo os algoritmos propostos, com algumas alterações e adições, ela é capaz de computar uma variedade grande de expressões numéricas diferentes de forma confiável.

O trabalho a princípio parece bastante simples, mas algumas dificuldades foram encontradas na hora de traduzir e computar as expressões. A decisão de criar uma nova lista para a tradução, ao invés de mudar a já existente, e separar cada número com espaços, por exemplo, foi tópico de bastante debate.

Em geral este código requer constância, isto é, convenções devem ser estabelecidas e seguidas. Considerando que o código foi escrito por dois alunos, nos trouxe um bom senso de trabalho em equipe, ensinou a importância de um código claro e bem comentado, e a necessidade de comunicação entre diferentes contribuidores de um projeto. Isso, sabemos, é de extrema importância para projetos futuros, sejam eles na universidade ou não.

O código final atende aos requerimentos e objetivos propostos em [1], e faz além ainda, mas acreditamos que outras modificações poderem ser feitas, adicionando novas funcionalidades. Estas ficarão para implementações futuras, no entanto. Por enquanto estamos satisfeitos com o algoritmo criado. A diversidade de testes realizados, como visto na seção 2.7, serve de prova para o bom funcionamento dele.

Entretanto não foi fácil chegar numa flexibilidade tamanha. Tivemos dificuldades principalmente com resultados negativos. Isso nos levou realmente a alterar todo o funcionamento da avaliação do programa para a maneira final. Outro aspecto complicado foi a divisão. A possibilidade de gerar um número fracionário, e mais, que este resultado poderia ser apenas um resultado parcial, foi bastante complicado. Não temos total certeza de que todas as possibilidades foram examinadas nesse sentido, mas resolvemos com sucesso todos os testes que realizamos (que foram bem mais extensos do que os testes mostrados na seção 2.7).

Os operadores unários também geraram alguma dor de cabeça. Eles precisaram ser tratados de forma diferente dos outros operadores, mas comportar corretamente estes operadores aumentaram imensamente a flexibilidade do algoritmo, então valeu a pena.

```
obrigado por usar a calculadora de expressoes ED
```

Referências

[1] Prof. Eduardo A. P. Alchieri. Estruturas de dados - trabalho 1. `http://www.cic.unb.br/~alchieri/disciplinas/graduacao/ed/pilha.pdf`, 2017. [Online].

[2] Autor Desconhecido. Notação infixa para pós-fixa. `http://www.vision.ime.usp.br/~pmiranda/mac122_2s14/aulas/aula13/aula13.html`, 2017. [Online].

A. Bibliotecas Locais

Devido ao tamanho das bibliotecas seguem links para acesso aos códigos.

linkedlist.h - `https://pastebin.com/fpeUFVLW`

stach.h - `https://pastebin.com/Yvszpvcj`

B. Exemplos de Expressões Infixas e Pós-fixas

Os prefixos ”in”e ”pós”referem-se, nas expressões, à posição dos operadores relativamente a seus operandos. Na notação infixa os operadores vão entre os operandos, e na notação pós-fixa (também conhecida como notação polonesa reversa), os operadores estão depois dos operandos. Observe a tabela 1. Note como na forma pós-fixa a próxima operação a ser realizada utiliza os dois números imediatamente antes do operador. Para mais informações ver [2].

Tabela 1. Exemplos de notação infixa e pós-fixa

infixa	pós-fixa
$a - b$	$ab-$
$a - b * c$	$abc * -$
$(a - b) * c$	$ab - c*$
$a + b * c^d - e$	$abcd^* + e-$
$a * (b + c) * (d - g) * h$	$abc + *dg - *h*$
$a * b - c * d^e/f + g * h$	$ab * cde^*f/ - gh * +$

C. Código Completo

Para acessar uma cópia completa do código e bibliotecas locais, bem como quaisquer atualizações feitas visite o repositório do GitHub ”calculadora-de-expressoes-ED”através deste link - `https://github.com/Gguidini/calculadora-de-expressoes-ED`.