Desenvolvimento de Calculadora para Expressões Lógicas Utilizando a Linguagem C

Fernando L. Eidt[1], **João P. Elger**[1], **João P. Pastório**[1]

[1]Engenharia de Computação – Universidade Tecnológica Federal do Paraná (UTFPR)
85.902-490 – Toledo – PR – Brazil

{nando--eidt, elger.jp, joaopedropastorio}@hotmail.com

Abstract. *This paper presents a brief explanation about the logic expressions, whose use is very important for a huge area of knowledge, especially for the computational area and for the discrete mathematics. The main focus of this paper is to show the development of a calculator using the C Programming Language for those logic expressions.*

Resumo. *Este artigo apresenta uma breve explanação sobre as expressões lógicas, cujo uso é muito importante para uma grande área do conhecimento, especialmente para a área computacional e para a matemática discreta. O principal foco deste artigo é mostrar o desenvolvimento de uma calculadora usando a Linguagem de Programação C para essas expressões lógicas.*

1. Introdução

Neste trabalho pretende-se apresentar uma introdução à lógica, com foco na lógica proposicional, através desta originam-se expressões lógicas que podem ser resolvidas e que geram um dentre dois resultados, verdadeiro ou falso. Para deter conhecimento no assunto de lógica proposicional, é necessário que o indivíduo tenha uma base de lógica clássica, como por exemplo a do livro *Organon* de Aristóteles (384 - 322 a.C.), este livro relata os intrumentos para se proceder corretamente no pensar [Silva Filho 2000].

Sendo a lógica proposicional uma das ferramentas fundamentais para a computação, este trabalho também visa apresentar uma calculadora de expressões lógicas, a qual fora desenvolvida utilizando a Linguagem de Programação C. Esta calculadora tem o objetivo de resolver expressões lógicas mostrando a respectiva tabela-verdade solução.

2. Lógica Proposicional

A Lógica Proposicional é uma forma mais simples de lógica, é a lógica utilizada por computadores durante seu funcionamento. Neste tipo de lógica, todos os fatos do mundo real são tratados como proposições [Martins 2012]. Proposição “vem de propor”que significa submeter à apreciação; requerer um juízo. Trata de uma sentença declarativa - algo que está declarado por meio de termos, palavras ou símbolos - e cujo conteúdo poderá ser considerado verdadeiro ou falso [de Lima 2005].

2.1. O Alfabeto da Lógica Proposicional

O alfabeto da Lógica Proposicional é constituido por símbolos de pontuação, símbolos de verdade, símbolos proposicionais e conectivos proposicional [de Souza 2008].

- símbolos de pontuação: (,);
- símbolos de verdade: true, false (1, 0);
- símbolos proposicionais: P, Q, R, S...;
- conectivos proposicionais: $\neg, \vee, \veebar, \wedge, \leftrightarrow, \rightarrow$.

Assim como na aritmética matemática normal, na lógica proposicional os símbolos proposicionais são usados como variáveis ou constantes, os conectivos proposicionais fazem o papel dos operadores aritméticos, os símbolos de verdade são os resultados que as proposições podem adotar e o símbolo de pontuação serve para separar argumentos em determinada sintaxe.

2.2. A Tabela Verdade

Para a análise de uma expressão de Lógica Proposicional ou proposições compostas, necessita-se avaliar todas as possibilidades de verdade das proposições que compõem a expressão. As possibilidades de interrelação entre as proposições de uma expressão lógica está diretamente ligadas pela expressão 2^n, onde *n* é o número de proposições na expressão. Por exemplo, caso temos um expressão com três proposições, P, Q, R, e onde S é a solução da interação entre as proposições, devemos criar a tabela verdade da seguinte forma:

Tabela 1. Uma tabela verdade 2^3 P, Q, R

P	Q	R	S
1	1	1	
1	1	0	
1	0	1	
1	0	0	
0	1	1	
0	1	0	
0	0	1	
0	0	0	

De forma à conseguirmos preencher a tabela verdade corretamente, devemos utilizar $\frac{2^n}{2}$ para cada proposição na expressão, para P nesse caso $\frac{2^3}{2} = 4$, logo deve-se preencher quatro elementos com um e quatro elementos com zero, para Q utilizaremos o resultado do preenchimento de P e realizaremos a mesma operação $\frac{4}{2} = 2$, logo deve-se preencher dois elementos com um e dois elementos com zero até alcançar 2^n, e assim por diante.

2.3. Os Conectivos Proposicionais ou Operadores Lógicos

Como já fora anteriormente citado na subseção 2.1., os operadores lógicos na lógica proposicional podem ser: $\neg, \vee, \veebar, \wedge, \leftrightarrow, \rightarrow$. Nesta subseção será mostrada a função de cada um desses operadores.

2.3.1. Operador de Negação

O operador de negação, simbolizado por $\neg$ ou $\sim$, tem o objetivo de inverter o valor lógico de determinada proposição. Como exemplo, mostraremos a tabela a seguir:

Tabela 2. Tabela verdade da negação de uma proposição qualquer

P	$\neg$P
1 (True)	0 (False)
0 (False)	1 (True)

2.3.2. Operador de Disjunção

O operador de disjunção, simbolizado por $\vee$ e também chamado de conectivo "ou", retorna valor verdadeiro (1) se, e somente se, pelo menos uma das proposições tiver valor verdadeiro, conforme tabela a seguir:

Tabela 3. Tabela verdade da disjunção entre duas proposições

P	Q	P$\vee$Q
1	1	1
1	0	1
0	1	1
0	0	0

2.3.3. Operador de Disjunção Exclusiva

O operador de disjunção exclusiva, também chamado de "ou... ou..."ou de "ou exclusivo"e simbolizado por $\veebar$, retorna valor verdadeiro (1) se, e somente se uma proposição tiver o valor falso e a outra tiver valor verdadeiro, conforme tabela a seguir:

Tabela 4. Tabela verdade da disjunção exclusiva entre duas proposições

P	Q	P$\veebar$Q
1	1	0
1	0	1
0	1	1
0	0	0

2.3.4. Operador de Conjunção

O operador de conjunção, simbolizado por $\wedge$ e também chamado de conectivo "e", retorna valor verdadeiro (1) se, e somente se, ambas as proposições tiverem valores verdadeiros,

conforme tabela a seguir:

Tabela 5. Tabela verdade da conjunção entre duas proposições

P	Q	P∧Q
1	1	1
1	0	0
0	1	0
0	0	0

2.3.5. Operador de Bicondição (Bicondicional)

O operador bicondicional, também chamado de conectivo "se e somente se"e simbolizado por $\leftrightarrow$, retorna valor verdadeiro (1) se, e somente se ambas proposições tiverem o mesmo valor lógico, conforme mostra a tabela a seguir:

Tabela 6. Tabela verdade da bicondição entre duas proposições

P	Q	P↔Q
1	1	1
1	0	0
0	1	0
0	0	1

2.3.6. Operador de Condição (Condicional)

O operador condicional, também chamado de conectivo "se... então"e simbolizado por $\rightarrow$, retorna valor falso (0) se, e somente se, a primeira proposição tiver valor falso e a segunda verdadeiro, conforme a tabela seguir nos mostra:

Tabela 7. Tabela verdade da condição entre duas proposições

P	Q	P→Q
1	1	1
1	0	0
0	1	1
0	0	1

2.4. Precedência dos Operadores

Os símbolos de pontuação (parênteses), assim como na aritmética normal, são empregados para priorizar um "cálculo proposicional". Esses símbolos podem ser omitidos quando os mesmos não alteram o resultado [Martins 2012]. Caso não haja o uso de parênteses, a seguir ordem de precedência deve ser adotada:

- $\neg$ = maior precedência;
- $\vee$ e $\wedge$ = precedência intermediária;
- $\rightarrow$ e $\leftrightarrow$ = menor precedência.

Além da precedência, ainda existem as regras de associatividade, que definem a prioridade no cálculo para conectivos de mesma precedência. Sendo essas:

- $\vee$ e $\wedge$ = conectivos associativos à esquerda;
- $\rightarrow$ e $\leftrightarrow$ = conectivos associativos à direita.

Como exemplo da regra de associação, considere as seguintes expressões lógicas:

- $P \vee Q \wedge R = (P \vee Q) \wedge R$
- $P \rightarrow Q \leftrightarrow R = P \rightarrow (Q \leftrightarrow R)$

3. Desenvolvimento da Calculadora para Cálculos Proposicionais

Através da base abordada pelo artigo sobre a Lógica Proposicional, agora poderemos explicar de maneira clara e precisa como fora o desenvolvimento do programa em Linguagem C que calcula expressões lógicas. O programa foi criado, testado e executado em uma plataforma Windows 64 bits e uma arquitetura Intel, as demais plataformas e arquiteturas não foram avaliadas com o código desenvolvido.

Este programa pode suportar até 26 proposições podendo ser essas todas as letras maiúsculas do alfabeto romano, ele irá receber a expressão de um arquivo *.txt* e irá retornar a tabela verdade solução dessa expressão em outro arquivo *.txt*.

3.1. Instruções para a Utilização do Programa

Para a utilização do programa, certas simbologias de alguns operadores lógicos foram substituídas, veja na tabela a seguir como são as entradas dos operadores nesta calculadora:

Tabela 8. Simbologias Adotadas no Programa

Simbologia Original	Simbologia no Programa
$\neg$	"$\sim$"
$\vee$	"v"
$\wedge$	"Acento Circunflexo"
$\rightarrow$	"->"
$\leftrightarrow$	"<->"
$\veebar$	"#"

A partir de agora, todas as expressões lógicas serão expressadas utilizando a simbologia do programa, com o intúito de tornar a explicação mais fácil e compreensível. Ademais, é importante ressaltar que o usuário só poderá calcular uma expressão lógica se utilizar as proposições lógicas em ordem alfabética, podendo essas serem posteriormente repetidas, exemplo:

- Av(B->C)<->B = Correto;
- Pv(B->C)<->B = Incorreto.

3.2. Lógica para a Criação do Programa

Durante o desenvolvimento da lógica do programa, a primeira etapa fora ver como poderíamos trabalhar com a expressão lógica de entrada de forma que o programador conseguisse manusea-la de maneira mais eficiente e precisa. Com isso em vista, fora decidido trabalhar individualmente com cada proposição da expressão de entrada, separando-as em uma string e criando uma matriz que trata-se cada linha como sendo uma proposição e cada coluna como sendo o valor da tabela verdade de cada proposição.

É importante lembrar que as linhas da matriz, posteriormente serão tratadas como proposições, logo a linha zero será o índice zero da string de proposições, representando, assim, a proposição A. O exemplo a seguir mostra como essa matriz se comportaria na memória a partir desta mesma lógica.

Tabela 9. Representação na matriz

	0	1	2	3
A = 0	1	1	0	0
B = 1	1	0	1	0

A segunda etapa de desenvolvimento da lógica se baseou em criar um laço de repetição com base no número de colunas da matriz de tabela verdade principal, substituindo os valores das proposições pelas suas respectivas posições de acordo com cada coluna. A seguir, veremos um exemplo de como a lógica iria operar por completo.

1. Recebe a expressão lógica de entrada de um arquivo *.txt*, seja esta expressão: (AvB);
2. Verifica quantas proposições existem na expressão lógica, no caso A e B;
3. Aloca a matriz com o número de linhas de proposições e o número de colunas de acordo com a expressão 2^n, sendo $n = 2$ no caso;
4. A matriz da tabela verdade é preenchida com valores através do trecho de código a seguir, neste trecho a variável *colunas* seria o tamanho da matriz de tabela verdade e a variável *Numero_De_Linhas* seria o número de proposições na expressão, as outras variáveis têm função de administrar e contar o preenchimento e todas as variáveis já estão incializadas no código completo.

```
1     for(i = 0; colunas%2 == 0 ; i++)
2     {
3         colunas = colunas/2;
4         contador_1 = 0;
5         contador_0 = 0;
6         for(j = 0, k = 0; j <Numero_De_Linhas; j++)
7         {
8             if(k < colunas)
9             {
10                Tabela_Verdade_Proposicoes[i][j] = '1';
11                contador_1++;
12            }
13            else
14            {
15                Tabela_Verdade_Proposicoes[i][j] = '0';
16                contador_0++;
17                if(contador_0 == contador_1)
```

```
18                      {
19                          k = 0;
20                          continue;
21                      }
22                  }
23                  k++;
24              }
25          }
```

5. Entra em um laço de repetição que substitui as proposições na expressão de acordo com seu valor na matriz de tabela verdade, a cada laço é verificada a operação entre as duas proposições. A operação é realizada e o símbolo da operação e substituído pelo resultado do cálculo proposicional, juntamente com a substituição dos valores por espaços. Exemplo para o caso (AvB):

Tabela 10. Tabela verdade de A e B

	0	1	2	3
A = 0	1	1	0	0
B = 1	1	0	1	0

Primeiro laço de repetição: (A v B) $\Rightarrow$ (1 v 1) $\Rightarrow$ (1);
Segundo laço de repetição: (A v B) $\Rightarrow$ (1 v 0) $\Rightarrow$ (1);
Terceiro laço de repetiçao: (A v B) $\Rightarrow$ (0 v 1) $\Rightarrow$ (1);
Quarto laço de repetição: (A v B) $\Rightarrow$ (0 v 0) $\Rightarrow$ (0).

6. Os valores originados pelo item 4. passam por uma filtragem que remove os parênteses e espaços da expressão;
7. Assim que os valores passarem pela filtragem, eles são salvos em um *array* de 2^n elementos, salvando-os em um arquivo de saída e sendo essa a resposta final.

3.3. Funções Chaves para a Execução do Programa

Neste tópico serão mostradas funções chaves do código para que o programa execute corretamente, ao todos temos 11 funções chaves (não contando a principal *main*) sendo essas:

- *void Vetor_Proposicoes*;
- *char *AlocarMatrizUnidimensional*;
- *char **AlocarMatrizBidimensional*;
- *char *Organiza_Matriz*;
- *int Contador_De_Caracteres*;
- *void Realiza_Negacao*;
- *void Realizar_Operacoes1*;
- *void Realizar_Operacoes2*;
- *char *Tira_Espaço*;
- *void Tira_Parenteses*;
- *Organiza_Expressão*.

A seguir, veremos a função e a explicação de cada uma dessas funções chaves, por favor, considere que as bibliotecas da Linguagem C *stdio.h*, *stdlib.h*, *math.h* e *string.h* estão incluídas em todas as funções e códigos aqui mostrados. Vale ressaltar que as funções não estão em ordem de importância ou ordem de funcionamento.

3.3.1. void Vetor_Proposicoes

Esta função tem o objetivo de analisar a expressão lógica de entrada e salvar todas as proposições como A, B, C, etc. e salvá-las numa string própria para que, futuramente, essa string seja utilizada como cabeçalho da tabela verdade resultante. Esta função recebe como parâmetro a *string* que é a expressão lógica de entrada e uma outra *string* que armazena as proposição da entrada.

Em C, uma *string* é uma série de caractéres terminada com um caractére nulo, representado por "\0" [do Lago Pereira 2010]. A Figura 1 a seguir, mostra de maneira didática como uma string inicializada como "verde e amarelo" se comporta na memória.

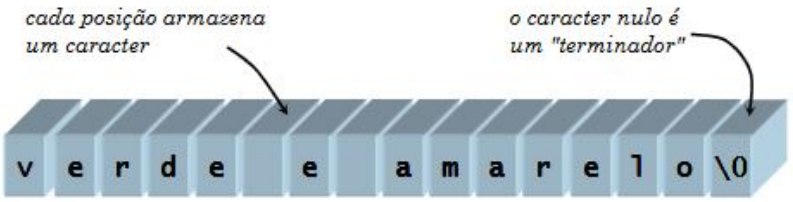

Figura 1. Exemplificação de uma string na memória

A seguir, mostra-se o código da função *void Vetor_Proposicoes*:

```
1  void Vetor_Proposicoes(char * expressao, char * proposicao)
2  {
3      int i, j, igual = 0, n = 0;
4      int Numero_de_caracteres = 0;
5
6      for(i = 0; expressao[i] != '\0'; i++)
7      {
8          if(expressao[i] >= 65 && expressao[i]<=90)
9          {
10             igual = 0;
11             for(j = 0; proposicao[j] != '\0'; j++)
12             {
13                 if(expressao[i] == proposicao[j])
14                     igual++;
15             }
16             if(igual == 0)
17             {
18             proposicao[n] = expressao[i];
19             n++;
20             }
21         }
22     }
23     proposicao[n] = '\0';
24 }
```

3.3.2. char *AlocarMatrizUnidimensional e char **AlocarMatrizBidimensional

Essas duas funções são primordias e baseiam seu funcionamento na mesma lógica, ambas tem a função de reservar um espaço na memória com determinada capacidade para

armazenar elementos do tipo *char*, sendo a unidimensional para uma *string* simples e a bidimensional para uma *string* de *strings*. As funções retornam um ponteiro (no caso da unidimensional) ou um duplo ponteiro (no caso da bidimensional), isso permite que o programador possa manipular as *strings* quando desejar.

É importante mencionar que, por estarmos utilizando a função *malloc*, para alocar espaço suficiente para as *strings*, os elementos não estão organizados de forma sequencial na memória conforme a Figura 1 nos mostra. A seguir, estão as funções *char *AlocarMatrizUnidimensional* e *char **AlocarMatrizBidimensional* utilizadas no programa:

```
1  char *AlocarMatrizUnidimensional(int Elementos)
2  {
3      char *NomeDaMatriz = malloc(Elementos*sizeof(char));
4
5      return NomeDaMatriz;
6  }
```

```
1  char **AlocarMatrizBidimensional(int Linhas, int Colunas)
2  {
3      int i = 0;
4
5      char **NomeDaMatriz = (char **) calloc(Linhas,sizeof(char*));
6
7      for(i = 0; i < Linhas; i++)
8      {
9          NomeDaMatriz[i]= (char*) calloc(Colunas,sizeof(char));
10     }
11
12     return NomeDaMatriz;
13 }
```

3.3.3. char Organiza_Matriz

A função *char Organiza_Matriz* tem como objetivo interpretar os operadores lógicos condicionais e bicondicionais, transformando o conectivo "se... então" para $>$ e o conectivo "se e somente se" para $<$. Essa transformação torna a expressão mais fácil de ser lida pelo compilador e mais fácil de ser trabalhada pelo programador. Após realizar seu objetivo, a função retorna um ponteiro que aponta para uma *string* com os elementos transformados e arrumados. A seguir, você estará vendo a função em questão.

```
1  char * Organiza_Matriz(int tamanho, char * expressao)
2  {
3      char * matriz_arrumada = malloc(tamanho*sizeof(char));
4      int i, j;
5      for(i = 0, j = 0; i < tamanho; i++, j++)
6      {
7          if(expressao[i] == '-')
8          {
9              i = i + 1;
10             matriz_arrumada[j] = '>';
11         }
12         if(expressao[i] == '<')
13         {
```

```
            i = i + 2;
            matriz_arrumada[j] = '<';
        }
        else
        {
            matriz_arrumada[j] = expressao[i];
        }
    }
        matriz_arrumada[j] = '\0';
    return matriz_arrumada;
}
```

3.3.4. int Contador_De_Caracteres

A função Contador_De_Caracteres tem o objetivo de contar quantas proposições existem na expressão de entrada, essa informação é necessária para a criação da tabela verdade da expressão lógica. Seu funcionamento se baseia em percorrer toda a expressão e verificar se existe um caractére maiúsculo nela. O código é dado a seguir.

```
int Contador_De_Caracteres(int TamanhoDaString, char *Termo)
{
    int i, j, contador, vezes_percoridas;
    int Numero_de_caracteres = 0;

    for(i =0; i < TamanhoDaString; i++)
    {
        if(Termo[i] >= 65 && Termo[i]<=90)
        {
            contador = 1;
            vezes_percoridas = 0;
            for(j = i; j>=0; j--)
            {
                if(Termo[i] != Termo[j])
                    contador++;

                vezes_percoridas++;
            }
            if(contador == vezes_percoridas)
                Numero_de_caracteres++;
        }
    }

    return Numero_de_caracteres;
}
```

3.3.5. void Realiza_Negacao

Dentre as operações lógicas mostradas anteriormente, a operação de negação é a primeira na ordem de precedência, com isso em mente desenvolvemos a função *void Realiza_Negacao*. No geral, ela tem o objetivo de verificar se a expressão lógica tem o símbolo de negação "∼" e, se constatado o símbolo, inverter o valor lógico da proposição. A seguir, veremos o código da função em questão.

```c
void Realiza_Negacao(char * expressao)
{
    int i;
    for(i = 0; expressao[i] != '\0'; i++)
    {
    if(expressao[i] == '~')
        {
            if(expressao[i+1] == '1')
            {
                expressao[i] = ' ';
                expressao[i+1] = '0';
            }
            else if(expressao[i+1] == '0')
            {
                expressao[i] = ' ';
                expressao[i+1] = '1';
            }
        }
    }
}
```

3.3.6. void Realizar_Operacoes1

Após a realização da negação pela função anteriormente mostrada, a função *void Realizar_Operacoes1* entra em ação, ela tem como objetivo realizar as operações de conjunção e disjunção entre proposições. Seu funcionamento é semelhante à função de realizar negação, porém aplicada para casos dos conectivo "ou" e "e". A seguir, contemple o código da função em questão.

```c
void Realizar_Operacoes1(char * expressao)
{
    int i, j;

    for(i = 0; expressao[i] != '\0'; i++)
    {
        if(expressao[i] == '0' || expressao[i] == '1')
        {
            if(expressao[i+2] == '0' || expressao[i+2] == '1')
            {
                if(expressao[i+1] == 'v')
                {
                    if(expressao[i] == '0' && expressao[i+2] == '0')
                    {
                        expressao[i] = ' ';
                        expressao[i+1] = '0';
                        expressao[i+2] = ' ';
                    }
                    if(expressao[i] == '1' && expressao[i+2] == '0')
                    {
                        expressao[i] = ' ';
                        expressao[i+1] = '1';
                        expressao[i+2] = ' ';
                    }
                    if(expressao[i] == '0' && expressao[i+2] == '1')
```

```
26                      {
27                          expressao[i] = ' ';
28                          expressao[i+1] = '1';
29                          expressao[i+2] = ' ';
30                      }
31                      if(expressao[i] == '1' && expressao[i+2] == '1')
32                      {
33                          expressao[i] = ' ';
34                          expressao[i+1] = '1';
35                          expressao[i+2] = ' ';
36                      }
37                  }
38                    if(expressao[i+1] == '^')
39                  {
40                      if(expressao[i] == '0' && expressao[i+2] == '0')
41                      {
42                          expressao[i] = ' ';
43                          expressao[i+1] = '0';
44                          expressao[i+2] = ' ';
45                      }
46                      if(expressao[i] == '1' && expressao[i+2] == '0')
47                      {
48                          expressao[i] = ' ';
49                          expressao[i+1] = '0';
50                          expressao[i+2] = ' ';
51                      }
52                      if(expressao[i] == '0' && expressao[i+2] == '1')
53                      {
54                          expressao[i] = ' ';
55                          expressao[i+1] = '0';
56                          expressao[i+2] = ' ';
57                      }
58                      if(expressao[i] == '1' && expressao[i+2] == '1')
59                      {
60                          expressao[i] = ' ';
61                          expressao[i+1] = '1';
62                          expressao[i+2] = ' ';
63                      }
64                  }
65              }
66          }
67          if(expressao[i] == '~')
68          {
69              if(expressao[i+1] == '1')
70              {
71                  expressao[i] = ' ';
72                  expressao[i+1] = '0';
73              }
74              else if(expressao[i+1] == '0')
75              {
76                  expressao[i] = ' ';
77                  expressao[i+1] = '1';
78              }
79          }
80      }
81  }
```

3.3.7. void Realizar_Operacoes2

Como últimas operações realizadas em operações lógicas, pela ordem de precedência, vem-nos os conectivos “se... então” e “se e somente se”. O funcionamento da função *void Realizar_Operacoes2* é equilavente à *void Realizar_Operacoes1*, porém com a lógica aplicada aos operadores lógicos de condição e de bicondição. A seguir, está destacado o código da função em questão.

```
1  void Realizar_Operacoes2(char * expressao, int tamanho)
2  {
3      int i, j;
4
5      for(i = 0; expressao[i] != '\0'; i++)
6      {
7          if(expressao[i] == '0' || expressao[i] == '1')
8          {
9              if(expressao[i+2] == '0' || expressao[i+2] == '1')
10             {
11                 if(expressao[i+1] == '>')
12                 {
13                     if(expressao[i] == '0' && expressao[i+2] == '0')
14                     {
15                         expressao[i] = ' ';
16                         expressao[i+1] = '1';
17                         expressao[i+2] = ' ';
18                     }
19                     if(expressao[i] == '1' && expressao[i+2] == '0')
20                     {
21                         expressao[i] = ' ';
22                         expressao[i+1] = '0';
23                         expressao[i+2] = ' ';
24                     }
25                     if(expressao[i] == '0' && expressao[i+2] == '1')
26                     {
27                         expressao[i] = ' ';
28                         expressao[i+1] = '1';
29                         expressao[i+2] = ' ';
30                     }
31                     if(expressao[i] == '1' && expressao[i+2] == '1')
32                     {
33                         expressao[i] = ' ';
34                         expressao[i+1] = '1';
35                         expressao[i+2] = ' ';
36                     }
37                 }
38
39                 if(expressao[i+1] == '<')
40                 {
41                     if(expressao[i] == '0' && expressao[i+2] == '0')
42                     {
43                         expressao[i] = ' ';
44                         expressao[i+1] = '1';
45                         expressao[i+2] = ' ';
46                     }
47                     if(expressao[i] == '1' && expressao[i+2] == '0')
```

```
48                      {
49                          expressao[i] = ' ';
50                          expressao[i+1] = '0';
51                          expressao[i+2] = ' ';
52                      }
53                      if(expressao[i] == '0' && expressao[i+2] == '1')
54                      {
55                          expressao[i] = ' ';
56                          expressao[i+1] = '0';
57                          expressao[i+2] = ' ';
58                      }
59                      if(expressao[i] == '1' && expressao[i+2] == '1')
60                      {
61                          expressao[i] = ' ';
62                          expressao[i+1] = '1';
63                          expressao[i+2] = ' ';
64                      }
65                  }
66
67                  if(expressao[i+1] == '#')
68                  {
69                      if(expressao[i] == '0' && expressao[i+2] == '0')
70                      {
71                          expressao[i] = ' ';
72                          expressao[i+1] = '0';
73                          expressao[i+2] = ' ';
74                      }
75                      if(expressao[i] == '1' && expressao[i+2] == '0')
76                      {
77                          expressao[i] = ' ';
78                          expressao[i+1] = '1';
79                          expressao[i+2] = ' ';
80                      }
81                      if(expressao[i] == '0' && expressao[i+2] == '1')
82                      {
83                          expressao[i] = ' ';
84                          expressao[i+1] = '1';
85                          expressao[i+2] = ' ';
86                      }
87                      if(expressao[i] == '1' && expressao[i+2] == '1')
88                      {
89                          expressao[i] = ' ';
90                          expressao[i+1] = '0';
91                          expressao[i+2] = ' ';
92                      }
93                  }
94              }
95          }
96
97      if(expressao[i] == '~')
98          {
99              if(expressao[i+1] == '1')
100             {
101                 expressao[i] = ' ';
102                 expressao[i+1] = '0';
103             }
```

```
104                 else if(expressao[i+1] == '0')
105                 {
106                     expressao[i] = ' ';
107                     expressao[i+1] = '1';
108                 }
109             }
110         }
111 }
```

3.3.8. char * Tira_Espaco

Esta função tem a simples função de retirar os espaços da expressão, retornando uma nova string sem os mesmos. Posteriormente, será mostrada a importância dessa função em nosso programa, haja vista que, ele se baseia em substituir determinados valores por espaços após operações. A seguir, o código da função *char * Tira_Espaço*.

```
1  char *Tira_Espaco(char * expressao, int tamanho)
2  {
3      int i = 0, j = 0;
4      char *strAUX = malloc(tamanho*sizeof(char));
5
6      for(i = 0; expressao[i] != '\0'; i++)
7      {
8          if(expressao[i] != ' ')
9          {
10             strAUX[j] = expressao[i];
11             j++;
12         }
13     }
14     strAUX[j] = '\0';
15     return strAUX;
16 }
```

3.3.9. void Tira_Parenteses

Esta função tem o objetivo de retirar os parênteses da prioridade de operações inseridos na expressão lógica de entrada após a realização da operação dentro desses parênteses. Sua funcionalidade se baseia em percorrer a expressão e, caso encontrar a abertura de um parênteses verificar se o elemento posterior à dois índices é a fecha desse parênteses, caso seja, ele elimina-os, pois só haverá uma proposição dentro e nenhuma operação a ser realizada. A seguir, o código da função.

```
1  void Tira_Parentes(char * expressao)
2  {
3      int i = 0;
4
5      for(i = 0; expressao[i] != '\0'; i++)
6      {
7          if(expressao[i] == '(')
8          {
9              if(expressao[i+2] == ')')
10             {
```

```
                    expressao[i] = ' ';
                    expressao[i+2] = ' ';
                }
            }
        }
}
```

3.3.10. void Organiza_Expressao

Essa função tem por objetivo verificar a matriz para preenchimento da expressão conforme a tabela verdade gerada. É importante ressaltar que na linha 9 desta função, o índice i da *string expressao* é compara com $j + 65$ por conta do valor da Tabela ASCII dos caractéres maiúsculos. A seguir, a função *void Organiza_expressão*:

```
1  void Organiza_Expressao(char * expressao, int tamanho, int n, char **
       Tabela_Verdade_Proposicoes)
2  {
3      int i, j;
4      for(i = 0; i < tamanho; i++)
5      {
6          if(expressao[i] >= 65 && expressao[i]<=90)
7          {
8              for(j = 0; j <= 25 ;j++)
9              if(expressao[i] == j+65)
10             expressao[i] = Tabela_Verdade_Proposicoes[j][n];
11         }
12     }
13 }
```

3.4. A Função *main*

A função *main* é a função chamada pelo sistema operacional para executar o programa, ela é considerada como a função principal, haja vista que a função *main* é a função que chama todas as outras anteriormente mostradas. A seguir, verifique o código dessa função, a qual irá se apropriar de todas as funções anteriormente citadas:

```
1  int main()
2  {
3      FILE * Arquivo_Entrada = fopen("entrada.txt","r");
4      FILE * Arquivo_Saida = fopen("saida.txt","w");
5      char expressao[255], expressao_original[255];
6      int Tamanho, colunas, i, j, k, n, s = 0;
7      int Numero_de_Caracteres, Numero_De_Linhas, contador_1 = 0,
       contador_0 = 0;
8      char * Nova_expressao, * saida, * proposicoes;
9      char **Tabela_Verdade_Proposicoes;
10
11     while(!feof(Arquivo_Entrada))
12     {
13         Numero_de_Caracteres = 0, Numero_De_Linhas = 0, contador_1 = 0,
       contador_0 = 0;
14         Tamanho = 0, colunas = 0, i = 0, j = 0, k = 0, n = 0, s = 0;
15
16
```

```
17      fgets(expressao,255,Arquivo_Entrada);
18
19      Tamanho = strlen(expressao);
20
21      for(i = 0;i <= Tamanho; i++)
22      {
23          if(expressao[i] == '\n')
24              expressao[i] = '\0';
25      }
26
27      Numero_de_Caracteres = Contador_De_Caracteres(Tamanho,&expressao);
28
29      proposicoes = AlocarMatrizUnidimensional(Numero_de_Caracteres);
30      Vetor_Proposicoes(expressao, proposicoes);
31
32
33      Numero_De_Linhas = pow(2,Numero_de_Caracteres);
34
35      colunas = Numero_De_Linhas;
36
37      for(i=0 ; expressao[i] != '\0'; i++)
38          expressao_original[i] = expressao[i];
39
40          expressao_original[i] = '\0';
41
42      Tabela_Verdade_Proposicoes = AlocarMatrizBidimensional(
    Numero_de_Caracteres, Numero_De_Linhas);
43
44
45      for(i = 0; colunas%2 == 0 ; i++) // preenche tabela verdade com 0
    ou 1
46      {
47          colunas = colunas/2;
48          contador_1 = 0;
49          contador_0 = 0;
50          for(j = 0, k = 0; j <Numero_De_Linhas; j++)
51          {
52
53              if(k < colunas)
54              {
55                  Tabela_Verdade_Proposicoes[i][j] = '1';
56                  contador_1++;
57              }
58              else
59              {
60                  Tabela_Verdade_Proposicoes[i][j] = '0';
61                  contador_0++;
62                  if(contador_0 == contador_1)
63                  {
64                      k = 0;
65                      continue;
66                  }
67              }
68              k++;
69          }
70      }
```

```
71
72
73      saida = AlocarMatrizUnidimensional(Numero_De_Linhas);
74
75
76      for(n = 0; n < Numero_De_Linhas; n++)
77      {
78
79      for(i=0 ; expressao[i] != '\0'; i++)
80          expressao[i] = expressao_original[i];
81
82          expressao[i] = '\0';
83
84          Tamanho = strlen(expressao);
85
86      Organiza_Expressao(expressao,Tamanho,n, Tabela_Verdade_Proposicoes)
        ;
87
88
89      Nova_expressao = organiza_matriz(Tamanho,expressao);
90
91
92      while(Tamanho >= 2)
93      {
94          Realiza_Negacao(Nova_expressao);
95          Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
96
97          Tira_Parentes(Nova_expressao);
98
99          Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
100
101         Realizar_Operacoes1(Nova_expressao);
102         Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
103
104         Tira_Parentes(Nova_expressao);
105
106         Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
107
108
109         Realizar_Operacoes2(Nova_expressao,Tamanho);
110         Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
111
112         Tira_Parentes(Nova_expressao);
113
114         Nova_expressao = Tira_Espaco(Nova_expressao, Tamanho);
115
116
117         Tamanho = strlen(Nova_expressao);
118
119
120     }
121
122
123     saida[s] = Nova_expressao[0];
124     s++;
125     }
```

```
126     saida[s] = '\0';
127
128
129     fprintf(Arquivo_Saida,"
        ------------------------------------------------------------------------------
        n");
130     fprintf(Arquivo_Saida,"Expressao: %s\n", expressao_original);
131     fprintf(Arquivo_Saida,"Solucao: \n\n");
132
133     for(j = 0; proposicoes[j] != '\0';j++)
134     {
135         fprintf(Arquivo_Saida,"%c | ",proposicoes[j]);
136     }
137     fprintf(Arquivo_Saida,"S\n");
138
139     for(j = 0; j < Numero_De_Linhas; j++)
140     {
141         for(i=0; i<Numero_de_Caracteres; i++)
142         {
143             fprintf(Arquivo_Saida,"%c | ",Tabela_Verdade_Proposicoes[i
        ][j]);
144         }
145         fprintf(Arquivo_Saida,"%c", saida[j]);
146         fprintf(Arquivo_Saida,"\n");
147     }
148     fprintf(Arquivo_Saida,"
        ------------------------------------------------------------------------------
        n");
149
150
151
152     }
153     fclose(Arquivo_Entrada);
154     fclose(Arquivo_Saida);
155
156     return 0;
157 }
```

Perceba que, como o programa é tratado através de uma variável do tipo *FILE*, é necessário que o usuário defina um arquivo *.txt* (Linha 3) para escrever as expressões lógicas, esse arquivo pode conter mais do que somente uma expressão lógica por vez, permitindo o usuário inserir uma expressão por linha. A saída (resultado) do programa também se dá em uma variável do tipo *FILE* (Linha 4), logo é esperado que o programa crie um arquivo *.txt* para mostrar a tabela verdade final.

4. Considerações Finais

Através deste artigo, conclui-se, portanto, que a calculadora de expressões lógicas opera e funciona para a maioria dos casos, porém com algumas limitações, sendo uma dessas, por exemplo, a necessidade de inserir as proposições por ordem alfabética. Todavia, vale ressaltar que o programa funcionou perfeitamente para todos os casos testados na plataforma em que fora desenvolvido.

Outrossim, destaca-se a importância da lógica proposicional em áreas que se relacionam à computação e à matemática discreta, nesse sentido percebe-se que utilizando o

programa desenvolvido neste artigo, é possível facilitar cálculos e implementar o mesmo para criação de diversos tipos de sistema, tais como Sistemas Inteligente (Inteligência Artificial).

Referências

de Lima, C. S. (2005). *Fundamentos de Lógica e Algorítmos*. IFRN - Instituto Federal de Educação, Ciência e Tecnologia do Rio Grande do Norte. Disponível em: <https://docente.ifrn.edu.br/cleonelima/disciplinas/fundamentos-de-programacao-2.8401.1m/fundamentos-de-logica-e-algoritmos-1.8401.1v/apostila-proposicoes-tabelas-verdade-conectivos-logicos>. Acesso em Junho, 2017.

de Souza, J. N. (2008). *Lógica para Ciência da Computação: Uma Introdução Concisa*. Elsevier-Campus, 2. edition.

do Lago Pereira, S. (2010). *Algoritmos e Lógica de Programação em C - Uma Abordagem Didática*, 1. edition.

Martins, L. G. A. (2012). *Fundamentos Básicos de Lógica Proposicional*. UFU - Universidade Federal de Uberlândia. Disponível em: <http://www.facom.ufu.br/∼gustavo/Logica/Apostila_LogicaProposicional.pdf>. Acesso em Junho, 2017.

Silva Filho, J. I. (2000). *Lógica Paraconsistente Anotada*. Emmy, Santos, SP, 1. edition.