

CES-11 Algoritmos e Estruturas de Dados

Laboratório 02 – 2o Período de 2020

Denis Loubach | dloubach@ita.br

1 Árvore Binária para Expressão Aritmética

O objetivo deste laboratório é montar uma árvore binária para calcular o valor resultante de uma determinada expressão aritmética. Isto é, processar a expressão, percorrendo sua respectiva árvore em profundidade.

Por exemplo, seja a expressão dada na Eq. (1).

$$\sin(4 \times (3.7 + \log_2 10.5)) + \frac{\cos(3.5 + \log_5 8)}{\sqrt{-(5)^2 + 4 \times 8 \times 2}} \quad (1)$$

Sua representação em formato de árvore pode ser ilustrada conforme a Fig. 1.

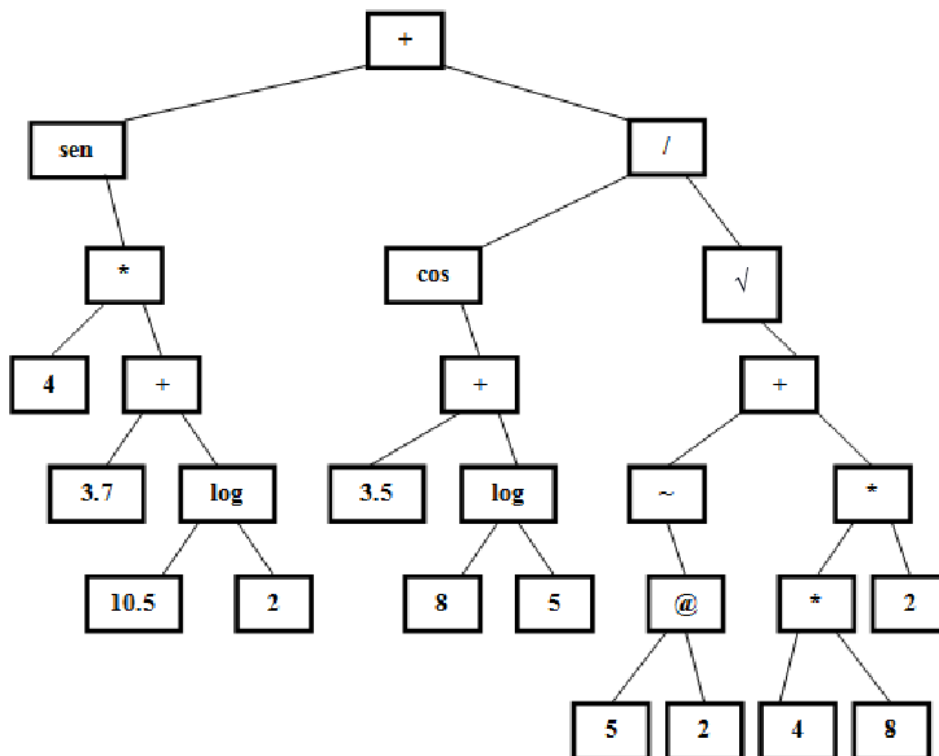


Figura 1: Representação da expressão Eq. (1) como árvore binária.

2 Algoritmo Base

O algoritmo a ser desenvolvido consiste em ler uma entrada, fazer a conversão para árvore binária e obter o resultado da expressão. Especificamente, é requisito implementar os seguintes passos:

1. Ler uma expressão na forma parentética;
2. Converter de parentética para polonesa;
3. Processar a polonesa para construir a árvore; e

4. Percorrer a árvore para o cálculo do valor resultante da expressão.

Esses passos são detalhados na sequência.

3 Leitura de uma Expressão na Forma Parentética

Esse passo do algoritmo consiste na digitação, ou leitura de um arquivo de texto plano, na forma parentética tal como o seguinte exemplo:

Listing 1: Exemplo de entrada de dados na forma parentética.

```
1 (((23 + 90) * (8 + (53 * 12))) + 42)
```

3.1 Orientação de Implementação

Obter os **átomos** contidos na *string* lida em forma parentética. Um **átomo** é um **elemento lógico** composto por um ou mais caracteres. Pode-se enumerá-los como:

1. número;
2. operadores; e
3. parêntesis.

Importante: todas as operações devem ser colocadas entre parêntesis.

3.1.1 Expressão

Seja uma expressão como a do tipo mostrado na Eq. (2).

$$\left(\left((n_1 + n_2) \times (n_3 + (n_4 \times n_5)) \right) + n_6 \right) \quad (2)$$

em que:

- n_i é um número inteiro não negativo com um ou mais dígitos decimais; e
- os operadores são **somente** soma ‘+’ e multiplicação ‘×’;

3.1.2 Implementação do Átomo

O átomo é uma estrutura contendo dois campos, conforme mostrado na Listagem 2.

1. tipo; e
2. atributo.

Listing 2: Declaração da estrutura do átomo.

```
1 // atom_st struct declaration
2 typedef struct{
3     type_et type;
4     attribute_ut attribute;
5 } atom_st;
```

O tipo (type) de um átomo, mostrado na Listagem 3, pode ser:

NUMBER : um número, quando o atributo for um número inteiro positivo;

OP : uma operação, quando o atributo for um dos caracteres para soma ‘+’ ou multiplicação ‘×’;

OPAR : um abre-parêntesis, quando o atributo for o caractere ‘(’;

CPAR : um fecha-parêntesis, quando o atributo for o caractere ')'; e

INVAL : inválido, quando o atributo for qualquer outro caractere não especificado.

Listing 3: Declaração das possibilidades de tipos do átomo.

```
1 // type_et declaration based on enum
2 typedef enum {NUMBER, OP, OPAR, CPAR, INVAL} type_et;
```

O eventual atributo de um átomo depende de seu tipo, a saber (Listagem 4):

NUMBER : o atributo é o valor, ou seja, o próprio número;

OP : o atributo é o próprio caractere '+' ou '*';

OPAR : o átomo não precisa de atributo;

CPAR : o átomo não precisa de atributo;

INVAL : o atributo é o próprio caractere.

Listing 4: Declaração das possibilidades do atributo do átomo.

```
1 // attribute_ut declaration based on union
2 typedef union{
3     int value;
4     char op;
5     char character;
6 } attribute_ut;
```

3.1.3 Esqueleto de Parte do Algoritmo

A seguir, pode ser visto o esqueleto de um algoritmo que lê a expressão na forma parentética, encontra seus átomos (tipos e atributos) e os guarda num vetor de átomos. Por final, escreve-se na tela o conteúdo desse vetor.

Num primeiro momento, a estratégia é ter um conjunto de variáveis globais, conforme mostrado na Listagem 5.

Listing 5: Listagem de variáveis globais.

```
1 #define MAX_ATOMS 100
2 #define MAX_EXPR 1000
3
4 //vetor de atomos para armazenar os atomos da expressao
5 atom_st parenthetical[MAX_ATOMS];
6
7 // vetor de caracteres com a expressao
8 typedef char expression_t[MAX_EXPR];
9 expression_t expr;
10
11 // numero de atomos em "parenthetical", e cursor para "expr"
12 int natoms, i;
13
14 // guarda um caracter de "expr"
15 char c;
```

Pode-se definir o protótipo das seguintes funções, conforme a Listagem 6:

Listing 6: Protótipos das funções.

```
1 // encontra todos os atomos de "expr" e os armazena em "parenthetical"
2 void make_atoms(void);
3
4 // escreve na tela o conteudo de todos os atomos num vetor de atomos
5 void print_atoms(atom_st[], int);
6
7 // procura o proximo caractere nao-branco em "expr"
8 char get_non_blank(void);
9
10 // retorna o proximo caractere de "expr"
11 char get_next(void);
12
13 // posiciona o cursor "i" no primeiro caractere de "expr"
14 void init_expr(void);
```

A Listagem 7 mostra um exemplo das chamadas de algumas funções.

Listing 7: Exemplo da chamada de algumas funções.

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <ctype.h>
4
5 #define MAX_ATOMS 100
6 #define MAX_EXPR 1000
7
8 typedef enum {NUMBER, OP, OPAR, CPAR, INVALID} type_et;
9
10 typedef union{
11     int value;
12     char op;
13     char character;
14 } attribute_ut;
15
16 typedef struct{
17     type_et type;
18     attribute_ut attribute;
19 } atom_st;
20
21 // global vars
22 typedef char expression_t[MAX_EXPR];
23
24 atom_st parenthetical_t[MAX_ATOMS];
25 expression_t expr;
26 int natoms, i;
27 char c;
28
29 void make_atoms(void);
30 void print_atoms(atom_st[], int);
31 char get_non_blank(void);
32 char get_next(void);
33 void init_expr(void);
34
35 int main(int argc, char *argv[])
36 {
```

```

37 printf("Expression: ");
38 fgets(expr, MAX_EXPR, stdin);
39 make_atoms();
40 printf("\nAtoms found in expression:\n");
41 print_atoms(parenthetical, natoms);
42
43 return 0;
44 }

```

A função `make_atoms` deverá seguir as seguintes orientações de implementação (Listagem 8):

Listing 8: Orientações de implementação para `make_atom`.

```

1 void make_atoms()
2 {
3     Zerar o numero de atomos encontrados;
4     Posicionar o cursor da expressao em seu inicio;
5     Procurar o primeiro caractere nao branco;
6     Enquanto esse caractere nao for o '\0'
7     {
8         Inserir um novo atomo em "parenthetical",
9         conforme o valor desse caractere;
10        - Caso seja digito
11        {
12            Coletar os outros digitos;
13            O tipo do atomo eh NUMBER;
14            O atributo eh o valor numerico da cadeia formada
15            pelos digitos coletados;
16        }
17        - Caso seja '+' ou '*'
18        {
19            O tipo do atomo eh OP;
20            O atributo eh o proprio caractere;
21        }
22        - Caso seja '(' ou ')'
23        {
24            O tipo do atomo eh respectivamente OPAR ou CPAR;
25            O atributo nao eh necessario;
26        }
27        - Caso seja qualquer outro caractere
28        {
29            O tipo do atomo eh INVALID;
30            O atributo eh o proprio caractere;
31        }
32        Incrementar natoms;
33        Procurar o proximo caractere nao branco;
34    }
35 }

```

Exemplo 3.1. Para a expressão `((12 + 32) * 13 & + 43;)`, a função `print_atoms` deverá gerar uma saída conforme ilustrado a seguir:

type		attribute
OPAR		
OPAR		
NUMBER		12
OP		+
NUMBER		32
CPAR		
OP		*

NUMBER		13
INVAL		&
OP		+
NUMBER		43
INVAL		;
CPAR		
CPAR		

□

Adicionalmente, um esboço de corpo das funções para percorrer a expressão, pode ser conforme mostrado na Listagem 9.

Listing 9: Esboço para o corpo das funções

```

1 char get_non_blank()
2 {
3     while (isspace(expr[i]) || (iscntrl(expr[i]) && expr[i] != '\0'))
4         i++;
5     return expr[i];
6 }
7
8 char get_next()
9 {
10    i++;
11    return expr[i];
12 }
13
14 void init_expr()
15 {
16    i = 0;
17 }

```

Ao completar o esqueleto desse passo do algoritmo base, pode-se verificar se um **vetor de átomos** guarda a **forma parentética de uma expressão**. Então, pode-se usar a seguinte definição recursiva:

1. Sendo m e n dois números inteiros não negativos com um ou mais dígitos cada, $(m + n)$ e $(m * n)$ são expressões na forma parentética.
2. Sendo α e β números inteiros não negativos ou expressões na forma parentética, $(\alpha + \beta)$ e $(\alpha * \beta)$ também são expressões nessa forma.

4 Conversão Forma Parentética para Notação Polonesa

Esse passo do algoritmo consiste na conversão da forma parentética para notação polonesa tal como o seguinte exemplo baseado na Listagem 1.

Listing 10: Exemplo de resultado de conversão da forma parentética para a notação polonesa.

```

1 23 90 + 8 53 12 * + * 42 +

```

4.1 Orientação de Implementação

Percorrer o **vetor de átomos**, construído no passo anterior, que guarda a **forma parentética de uma expressão** e, ao encontrar um:

- número, inseri-lo no final de uma **lista com estrutura encadeada** dedicada para guardar a notação polonesa;

- operador, colocá-lo numa **pilha**;
- abre-parêntesis, nada fazer;
- fecha-parêntesis, desempilhar um operador da pilha e colocá-lo no final da lista com estrutura encadeada dedicada para guardar a notação polonesa.

4.1.1 TAD Pilha

Implementar o TAD pilha conforme as seguintes declarações e protótipos.

Listing 11: Arquivo .h

```

1 /*
2  * Abstract Data Type Declarations -- STACK
3  * author: dloubach
4  *
5  */
6 #ifndef ADT_STACK_H
7 #define ADT_STACK_H
8
9 #include <stdbool.h>
10
11 /* ADT stack type declaration */
12 typedef struct node_st* stack_t;
13
14 /* ADT stack operators */
15 /* inicializa uma pilha vazia */
16 void adt_initStack(stack_t*);
17
18 /* adiciona um elemento no topo da pilha*/
19 void adt_pushStack(stack_t*, atom_st);
20
21 /* remove um elemento do topo da pilha */
22 void adt_popStack(stack_t*);
23
24 /* retorna o elemento do topo da pilha sem remove-lo */
25 atom_st adt_topStack(stack_t);
26
27 /* retorna <true> quando a pilha encontra-se vazia */
28 bool adt_isEmptyStack(stack_t);
29
30 /* esvazia a pilha */
31 void adt_emptyStack(stack_t*);
32
33 #endif

```

Listing 12: Implementação, arquivo .c

```

1  /*
2  * Abstract Data Type Implementations -- STACK
3  * author: dloubach
4  *
5  */
6
7  #include <stdlib.h>
8  #include "adt_stack.h"
9
10 /* ADT stack node implementation */
11 struct node_st{
12     atom_st element;
13     struct node_st* next;
14 };
15
16
17 /* ADT stack operators */
18
19 /* inicializa uma pilha vazia */
20 void adt_initStack(stack_t* stackArg)
21 {
22     ...
23 }
24
25 /* adiciona um elemento no topo da pilha*/
26 void adt_pushStack(stack_t* stackArg, int elementArg)
27 {
28     ...
29 }
30
31 /* remove um elemento do topo da pilha */
32 void adt_popStack(stack_t* stackArg)
33 {
34     ...
35 }
36
37 /* retorna o elemento do topo da pilha sem remove-lo */
38 int adt_topStack(stack_t stackArg)
39 {
40     ...
41 }
42
43 /* retorna <true> quando a pilha encontra-se vazia */
44 bool adt_isEmptyStack(stack_t stackArg)
45 {
46     ...
47 }

```

4.1.2 TAD Lista Encadeada

Implementar o TAD Lista com estrutura encadeada. Poderá ser re-utilizado todo o código implementado no laboratório TAD-lista-conjunto, agora substituindo o tipo do elemento para `atom_st`.

5 Construção da Árvore Binária com base na Notação Polonesa

O TAD para árvore pode ter as seguintes declarações:

```
1 typedef struct cell {
2     atom_st *elem;
3     struct cell *parent, *lchild, *rchild; //parent, left & right childs
4 } cell_st;
5
6 typedef cell_st *bintree_node;
7 typedef cell_st *bintree_t;
8
9 bintree_t bt;
```

A definição e implementação da lista de operadores do TAD Árvore Binária deve ser realizado conforme visto nas aulas teóricas.

6 Realizar uma Busca em Profundidade na Árvore Binária para Efetuar o Cálculo da Expressão Matemática

Imprima a forma parentética original da expressão a partir árvore binária.

6.1 Orientação de Implementação

Percorrer a árvore binária em **ordem-central**, escrevendo os atributos dos nós e escrevendo parêntesis em momentos necessários.

O cálculo do valor da expressão na árvore binária pode ser realizado tal como mostrado na Listagem 13.

Listing 13: Cálculo do valor da expressão.

```
1 int eval_expression(bintree_t bt)
2 {
3     caso a raiz de A seja:
4     {
5         - numero: retornar o valor do numero;
6         - operador:
7             caso o operador seja:
8             {
9                 // soma
10                 '+': retornar eval_expression(bt->lchild) +
11                     eval_expression(bt->rchild);
12                 // multiplicacao
13                 '*': retornar eval_expression(bt->lchild) *
14                     eval_expression(bt->rchild);
15             }
16     }
17 }
```

7 Entregas

A entrega deverá ser realizada conforme enunciado no *Google Classroom*.

Referências

- [1] F. C. Mokarzel, “Apostila de ces-11 estruturas de dados em slides.” ITA, 2011.
- [2] F. C. Mokarzel, “Roteiro para aulas práticas em slides.” ITA, 2011.