Instituto Tecnológico de Aeronáutica - ITA
Divisão de Ciência da Computação - IEC

CES-11 Algoritmos e Estruturas de Dados
Laboratório EXAME - parte 1 – 2o Período de 2021
Cesar Marcondes | cmarcondes@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).

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

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

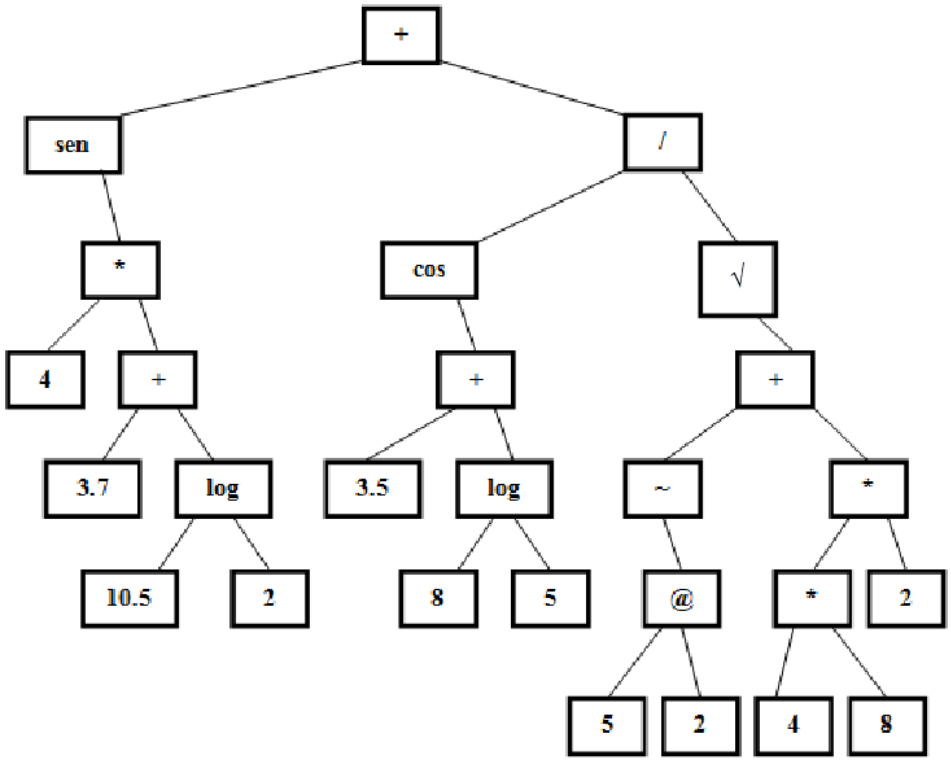

Figura 1: Representação da expressão Eq. (1) como árvore binária.

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.

```
(((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).

$$\Big(\big((n_1 + n_2) \times \big(n_3 + (n_4 \times n_5)\big)\big) + n_6\Big) \tag{2}$$

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.

```
// atom_st struct declaration
typedef struct{
  type_et type;
  attribute_ut attribute;
} 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.

```
// type_et declaration based on enum
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.

```
// attribute_ut declaration based on union
typedef union{
  int value;
  char op;
  char character;
} 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.

```
#define MAX_ATOMS 100
#define MAX_EXPR 1000

//vetor de atomos para armazenar os atomos da expressao
atom_st parenthetical[MAX_ATOMS];

// vetor de caracteres com a expressao
typedef char expression_t[MAX_EXPR];
expression_t expr;

// numero de atomos em "parenthetical", e cursor para "expr"
int natoms, i;

// guarda um caracter de "expr"
char c;
```

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

Listing 6: Protótipos das funções.

```
// encontra todos os atomos de "expr" e os armazena em "parenthetical"
void make_atoms(void);

// escreve na tela o conteudo de todos os atomos num vetor de atomos
void print_atoms(atom_st[], int);

// procura o proximo caractere nao-branco em "expr"
char get_non_blank(void);

// retorna o proximo caractere de "expr"
char get_next(void);

// posiciona o cursor "i" no primeiro caractere de "expr"
void init_expr(void);
```

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

Listing 7: Exemplo da chamada de algumas funções.

```
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>

#define MAX_ATOMS 100
#define MAX_EXPR 1000

typedef enum {NUMBER, OP, OPAR, CPAR, INVAL} type_et;

typedef union{
    int value;
    char op;
    char character;
} attribute_ut;

typedef struct{
  type_et type;
  attribute_ut attribute;
} atom_st;

// global vars
typedef char expression_t[MAX_EXPR];

atom_st parenthetical_t[MAX_ATOMS];
expression_t expr;
int natoms, i;
char c;

void make_atoms(void);
void print_atoms(atom_st[], int);
char get_non_blank(void);
char get_next(void);
void init_expr(void);

int main(int argc, char *argv[])
{
```

```
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 INVAL;
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**, construido 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

```
/*
 * Abstract Data Type Declarations -- STACK
 * author: dloubach
 *
 */
#ifndef ADT_STACK_H
#define ADT_STACK_H

#include <stdbool.h>

/* ADT stack type declaration */
typedef struct node_st* stack_t;

/* ADT stack operators */
/* inicializa uma pilha vazia */
void adt_initStack(stack_t*);

/* adiciona um elemento no topo da pilha*/
void adt_pushStack(stack_t*, atom_st);

/* remove um elemento do topo da pilha */
void adt_popStack(stack_t*);

/* retorna o elemento do topo da pilha sem remove-lo */
atom_st adt_topStack(stack_t);

/* retorna <true> quando a pilha encontra-se vazia */
bool adt_isEmptyStack(stack_t);

/* esvazia a pilha */
void adt_emptyStack(stack_t*);

#endif
```

Listing 12: Implementação, arquivo .c

```c
/*
 * Abstract Data Type Implementations -- STACK
 * author: dloubach
 *
 */

#include <stdlib.h>
#include "adt_stack.h"

/* ADT stack node implementation */
struct node_st{
  atom_st element;
  struct node_st* next;
};


/* ADT stack operators */

/* inicializa uma pilha vazia */
void adt_initStack(stack_t* stackArg)
{
...
}

/* adiciona um elemento no topo da pilha*/
void adt_pushStack(stack_t* stackArg, int elementArg)
{
...
}

/* remove um elemento do topo da pilha */
void adt_popStack(stack_t* stackArg)
{
...
}

/* retorna o elemento do topo da pilha sem remove-lo */
int adt_topStack(stack_t stackArg)
{
...
}

/* retorna <true> quando a pilha encontra-se vazia */
bool adt_isEmptyStack(stack_t stackArg)
{
...
}
```

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:

```
typedef struct cell {
  atom_st *elem;
  struct cell *parent, *lchild, *rchild; //parent, left & right childs
} cell_st;

typedef cell_st *bintree_node;
typedef cell_st *bintree_t;

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.

```
int eval_expression(bintree_t bt)
{
  caso a raiz de A seja:
  {
    - numero: retornar o valor do numero;
    - operador:
        caso o operador seja:
        {
          // soma
          '+': retornar eval_expression(bt->lchild) +
                        eval_expression(bt->rchild);
          // multiplicacao
          '*': retornar eval_expression(bt->lchild) *
                        eval_expression(bt->rchild);
        }
  }
}
```

7 Entregas

A entrega deverá ser realizada conforme enunciado no *run.codes*.

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.