**UNIVERSIDADE FEDERAL DE ALAGOAS**

**INSTITUTO DE COMPUTAÇÃO**

**CIÊNCIA DA COMPUTAÇÃO**

**VITOR BARCELOS DE CERQUEIRA**

**PEDRO MATEUS VERAS SIMÕES**

**Meer Compiler - Mais um compilador**

**Maceió - Alagoas**

**2021**

VITOR BARCELOS DE CERQUEIRA

PEDRO MATEUS VERAS SIMÕES

**Meer Compiler - Mais um compilador**

Trabalho apresentado à disciplina Compiladores, pelo Curso de Ciência da Computação da Universidade Federal de Alagoas - UFAL, ministrada pelo professor Alcino Dall'Igna Júnior.

Maceió - Alagoas

2021

**SUMÁRIO**

[**1 ESTRUTURA GERAL DE UM PROGRAMA**](#_3znysh7) **3**

[**2 ESPECIFICAÇÃO DE TIPOS DE DADOS**](#_eau323xyoxy2) **3**

[2.1 Booleano](#_iavwz35s3obl) 3

[2.2 Caractere](#_4jlce6y8ansx) 3

[2.3 Cadeia de Caracteres (String)](#_qnl9qw49bxaz) 4

[2.4 Inteiro](#_3cdxb5phe0cx) 4

[2.5 Ponto Flutuante](#_fphwtbc2rfhr) 4

[2.6 Arranjos unidimensionais](#_jxvqdh130286) 5

[**3 OPERAÇÕES DE TIPO DE DADOS**](#_4r287zv91mq1) **5**

[3.1 Operações](#_yuoas5e082f4) 5

[3.2 Coerções Explícitas](#_ggitem559rng) 6

[3.3 Coerções Implícitas](#_c046zy9fg8s) 7

[3.4 Precedência e Associatividade (Aritméticos)](#_8hiwvlbp8h1p) 7

[3.5 Precedência e Associatividade (Lógicos)](#_dw3u0w2p0p59) 7

[3.6 Precedência e Associatividade (Relacionais)](#_q2ufx8hr3pen) 8

[3.7 Precedência e Associatividade (Igualdade)](#_49eim1bkaa4i) 8

[3.8 Precedência e Associatividade (Todos)](#_9l28wi7m43qp) 8

[**4 INSTRUÇÕES**](#_aaynhfa7mez2) **9**

[4.1 Estrutura condicional](#_x6an9c65oll6) 9

[4.2 Estrutura iterativa com controle lógico](#_6mouiskto8as) 10

[4.3 Estrutura iterativa controlada por contador](#_o5t3i5ebag9r) 10

[4.4 Entrada](#_o4bueeldasnt) 10

[4.5 Saída](#_mhz8jhp643pn) 11

[4.6 Atribuição](#_o77vcmx1tr2q) 11

[**5 FUNÇÕES**](#_efnj1i2zf6u4) **11**

[**6 EXEMPLOS DE PROGRAMA**](#_o1y2owmux0p1) **12**

[6.1 Hello world](#_ddgm3p6cyblf) 12

[6.2 Shell sort](#_wg22psgmtwzn) 12

[6.3 Fibonacci](#_pvuuksd70c2h) 13

[**REFERÊNCIAS**](#_ai6hli90874u) **14**

# 1 ESTRUTURA GERAL DE UM PROGRAMA

Para definir uma estrutura de um programa criado na linguagem Meer que fosse simples de implementar foi definido, portanto, que: as funções não podem ser declaradas dentro de outras, isto é, não é permitido que exista qualquer nível de **aninhamento** entre elas. Nesse caso, as declarações são somente válidas no **escopo global** do programa em questão; as instruções e as variáveis só podem ser declaradas no **escopo local** das funções, nunca fora dele e, para finalizar, o **ponto de partida** de um programa qualquer sempre será a execução da função identificada com a **palavra reservada “init”**. Essa função de inicialização precisa, **obrigatoriamente**, ter a assinatura a seguir, com o retorno 0 (zero) em caso de sucesso na execução do programa, ou diferente de 0 (zero) em caso de falha.



| def init(var args: string[]): **int** begin  **return** 0; end |
| --- |

# 

# 2 ESPECIFICAÇÃO DE TIPOS DE DADOS

### 2.1 Booleano

Para identificar uma variável ou um retorno de função do tipo booleano é preciso usar a palavra reservada **“boolean”**. As palavras reservadas **“true”** e **“false”** são usadas como constantes literais.

| def booleanWorkspace(): **boolean** begin  var anyBool: boolean = true;  **return** anyBool; end |
| --- |

### 2.2 Caractere

Para identificar uma variável do tipo caractere é preciso usar a palavra reservada **“char”**. Esse tipo de dados suporta qualquer caractere da tabela **ASCII**. A constante literal desse tipo é um caractere entre aspas simples, ex. **‘@’**.

| def charWorkspace(): **char** begin  var anyChar: char = '@';  **return** anyChar; end |
| --- |

### 2.3 Cadeia de Caracteres (String)

Para identificar uma variável do tipo cadeia de caracteres é preciso usar a palavra reservada **“string”**. Esse tipo de dados suporta qualquer cadeia de caracteres da tabela **ASCII**. A constante literal desse tipo é uma cadeia de caracteres entre aspas duplas, ex. **“@#$a1”**. Uma vez que a memória disponível para um programa qualquer depende também de quanto o Sistema Operacional disponibiliza, então, o limite de caracteres de uma string na linguagem Meer não será fixo. Esse limite vai depender de quanta memória disponível existe para um programa usar. Por fim, o **“\n”** será o sinalizador do final de uma string.

| def stringWorkspace(): **string** begin  var anyString: string = “@#$a1”;  **return** anyString; end |
| --- |

### 2.4 Inteiro

Para identificar uma variável do tipo inteiro é preciso usar a palavra reservada **“int”**. Esse tipo de dados suporta qualquer número inteiro de 64 bits. A constante literal desse tipo é uma sequência de dígitos, ex. **123**.

| def intWorkspace(): **int** begin  var anyInt: int = 123;  **return** anyInt; end |
| --- |

### 2.5 Ponto Flutuante

Para identificar uma variável do tipo ponto flutuante é preciso usar a palavra reservada **“float”**. Esse tipo de dados suporta qualquer número de ponto flutuante de 64 bits. A constante literal desse tipo é uma sequência de dígitos, seguido por um ‘.’ e seguido por outra sequência de dígitos, ex. **12.34**.

| def floatWorkspace(): **float** begin  var anyFloat: float = 12.34;  **return** anyFloat; end |
| --- |

### 2.6 Arranjos unidimensionais

Para definir um arranjo unidimensional é preciso usar os colchetes **“[]”** após a palavra reservada do tipo, ex. **int[]**. Caso os colchetes estejam vazios, o tamanho do arranjo será dinâmico, caso seja definido uma constante literal do tipo inteiro entre os colchetes, então o arranjo terá tamanho fixo igual a essa constante. Além disso, é possível usar todos os tipos de dados da linguagem para definir arranjos, exceto o tipo **“void”** que é utilizado apenas no contexto de retorno de função.

| def arrayWorkspace(): int**[]** begin  var anyArray: int[3] = [1, 2, 3];  **return** anyArray; end |
| --- |

# 3 OPERAÇÕES DE TIPO DE DADOS

### 3.1 Operações

Na tabela a seguir é especificado quais operações cada tipo de dados suporta. O tipo de dados “char” possui uma **operação de sobrecarga** no operador aritmético “+”. O resultado da concatenação de dois, ou mais, caracteres vai ser uma “string”. Isso acontece porque o tamanho máximo do tipo “char” é 1 (um). Tanto os operadores relacionais do “char” quanto da “string” são baseados na tabela **ASCII**. Na prática, comparar dois caracteres, por exemplo, é comparar as devidas posições nessa tabela. A mesma situação vai acontecer na “string”, com todos os caracteres sendo comparados, se assim for necessário.

| **TIPO DE DADOS** | **OPERAÇÕES** |
| --- | --- |
| int | **Aritméticos:** +, -, \*, /, +(unário), -(unário)  **Relacionais:** <, >, <=, >=  **Igualdade:** ==, != |
| float | **Aritméticos:** +, -, \*, /, +(unário), -(unário)  **Relacionais:** <, >, <=, >=  **Igualdade:** ==, != |
| char | **Relacionais:** <, >, <=, >= (**ASCII**)  **Sobrecarga:** +(concatenação): **string**  **Igualdade:** ==, != (**ASCII**) |
| string | **Relacionais:** <, >, <=, >= (**ASCII**)  **Sobrecarga:** +(concatenação): **string**  **Igualdade:** ==, != (**ASCII**) |
| boolean | **Igualdade:** ==, !=  **Lógicos:** and, or, not |

### 3.2 Coerções Explícitas

As coerções explícitas podem ser feitas por meio das funções nativas: **Int()**, **Float()**, **Char()**, **String()** e **Boolean()**. Nas quais os parâmetros de entrada serão de um tipo de dados e o retorno será o valor convertido. Na tabela a seguir é especificado quais tipo de dados cada função suporta.

| **FUNÇÃO** | **TIPOS SUPORTADOS** |
| --- | --- |
| Int() | boolean (false → 0 e true → 1)  float (5.6 → 5 (Redução))  string (“10” → 10)  char ('@' → 64) |
| Float() | string (“10.0” → 10.0 e “10” → 10.0)  boolean (false → 0.0 e true → 1.0)  char ('@' → 64.0)  int (5 → 5.0) |
| Char() | float (64.0 → '@')  int (64 → '@') |
| String() | boolean (false → “false” e true → “true”)  float (5.0 → “5.0”)  char ('@' → “@”)  int (5 → “5”) |
| Boolean() | string (' ' (vazia) → false e “a..z” → true)  float (0.0 → false e ≠ de 0.0 → true)  int (0 → false e ≠ de 0 → true)  char ('@' → true) |

### 3.3 Coerções Implícitas

As coerções implícitas podem ser feitas através de operações entre os diferentes tipos de dados suportados pela linguagem. Na tabela a seguir é especificado quais coerções podem ser realizadas de modo implícito.

| **OPERAÇÕES** | **RESULTADOS** |
| --- | --- |
| (int | float | boolean | char | string) + (string | char) | string |
| (int + float) | float |
| (int + boolean) | int |
| (float + boolean) | float |
|  |  |

### 3.4 Precedência e Associatividade (Aritméticos)

A tabela a seguir mostra a precedência e associatividade dos operadores aritméticos da linguagem, em ordem decrescente de precedência (da maior para a menor).

| **OPERADOR** | **DESCRIÇÃO** | **ASSOCIATIVIDADE** |
| --- | --- | --- |
| () | parênteses | da esquerda para a direita |
| +, - | positivo e negativo unário | da direita para a esquerda |
| \*, / | multiplicação e divisão | da esquerda para a direita |
| +, - | soma e subtração | da esquerda para a direita |

### 3.5 Precedência e Associatividade (Lógicos)

A tabela a seguir mostra a precedência e associatividade dos operadores lógicos da linguagem, em ordem decrescente de precedência (da maior para a menor).

| **OPERADOR** | **DESCRIÇÃO** | **ASSOCIATIVIDADE** |
| --- | --- | --- |
| not | negação lógica | da direita para a esquerda |
| and | E lógico | da esquerda para a direita |
| or | OR lógico | da esquerda para a direita |
|  |  |  |

### 3.6 Precedência e Associatividade (Relacionais)

A tabela a seguir mostra a precedência e associatividade dos operadores relacionais da linguagem, em ordem decrescente de precedência (da maior para a menor).

| **OPERADOR** | **DESCRIÇÃO** | **ASSOCIATIVIDADE** |
| --- | --- | --- |
| <  >  <=  >= | menor que,  maior que,  menor ou igual a,  maior ou igual a | da esquerda para a direita |

### 3.7 Precedência e Associatividade (Igualdade)

A tabela a seguir mostra a precedência e associatividade dos operadores relacionais da linguagem, em ordem decrescente de precedência (da maior para a menor).

| **OPERADOR** | **DESCRIÇÃO** | **ASSOCIATIVIDADE** |
| --- | --- | --- |
| ==  != | igual a,  diferente a, | da esquerda para a direita |

### 3.8 Precedência e Associatividade (Todos)

A tabela a seguir mostra a precedência e associatividade de todos os operadores da linguagem, em ordem decrescente de precedência (da maior para a menor).

| **OPERADOR** | **DESCRIÇÃO** | **ASSOCIATIVIDADE** |
| --- | --- | --- |
| () | parênteses | da esquerda para a direita |
| +, - | positivo e negativo unário | da direita para a esquerda |
| \*, / | multiplicação e divisão | da esquerda para a direita |
| +, - | soma e subtração | da esquerda para a direita |
| <  >  <=  >= | menor que,  maior que,  menor ou igual a,  maior ou igual a | da esquerda para a direita |
| ==  != | igual a,  diferente a, | da esquerda para a direita |
| not | negação lógica | da direita para a esquerda |
| and | E lógico | da esquerda para a direita |
| or | OR lógico | da esquerda para a direita |

# 4 INSTRUÇÕES

Toda instrução de linha precisa de um **;** para terminar a instrução. Um bloco de instrução é definido pelas palavras reservadas **‘begin’** e **‘end’**.

### 4.1 Estrutura condicional

São definidas pela palavra reservada ‘**if’**, em seguida uma expressão lógica entre parênteses, que caso retorne verdadeiro o bloco da condicional **if** será executado. Caso o valor da expressão seja falso, a próxima estrutura condicional será executada, que será um **elseif** ou um **else**.

O **elseif** é similar ao **if**, sendo usado apenas nas estruturas subsequentes do **if**. O **else** será executado caso todas as estruturas condicionais anteriores tenham sido falsas.

| if (a > b)  begin  ...  end  elseif (b > c)  begin  ...  end  else  begin  ...  end |
| --- |
|  |

### 4.2 Estrutura iterativa com controle lógico

É definida usando a palavra reservada **‘while’**, em seguida a expressão lógica entre parênteses a ser testada e depois seu bloco de instruções. Essa estrutura é pré-teste.

| while (true)  begin  ... end |
| --- |

### 4.3 Estrutura iterativa controlada por contador

É definida usando a palavra reservada ‘**for’**, em seguida três parâmetros são passados entre parênteses e separados por ‘**,’**. O primeiro parâmetro é a variável que será iterada, o segundo a variável ou constante literal com o limite da iteração e o terceiro e único opcional, o passo da iteração, que pode ser uma variável ou constante literal. Caso o passo não seja dado, seu valor será 1.

| for (var i: int = 0, 10, 2)  begin  ... end |
| --- |

### 4.4 Entrada

Definido pela palavra reservada **‘input’**, em seguida as variáveis que receberão os valores lidos, separados por **‘,’** e entre parênteses. Uma formatação pode ser especificada entre aspas dupla e a esquerda das variáveis/constantes literais.

| var a: int, b: int;  input(“%d:%d”, a, b); |
| --- |

### 4.5 Saída

Definido pela palavra reservada **‘print’**, em seguida as variáveis ou constantes literais que serão colocadas na tela, separados por **‘,’** e entre parênteses. Uma formatação pode ser especificada entre aspas dupla e a esquerda das variáveis/constantes literais.

| var a: int, b: int = 1, 2;  print(“%d:%d”, a, b); |
| --- |

### 

### 4.6 Atribuição

Definido pelo comando **‘=’**, a esquerda é necessário uma ou mais variáveis, e na direita uma ou mais (variáveis ou constantes literais). É necessário que o número de elementos à esquerda seja exatamente igual ao número de elementos à direita. Os elementos são separados por **‘,’**.

| var a: int, b: int = 1, 2; |
| --- |

# 5 FUNÇÕES

Definida pela palavra reservada **‘def’**, em seguida o identificador da função e a lista de parâmetros. O tipo de retorno da função é definido usando o símbolo **‘:’**. O bloco do escopo da função é definido pelas palavras reservadas **‘begin’** e **‘end’**. A palavra reservada **‘return’** é usada para definir o retorno da função. Caso o tipo da função seja **void**, ela não terá retorno. Os parâmetros são sempre passados por Valor-Cópia.

| def name(var a: int, var b: int): **int** begin  ...  **return** a; end |
| --- |

# 6 EXEMPLOS DE PROGRAMA

### 6.1 Hello world

| def init(var args: string[]): **int** begin  print(“Hello World”);  **return** 0;  end |
| --- |

### 6.2 Shell sort

| def shellsort(var list: int[], var size: int): **int[]**  begin  var gap: int = Int(size / 2);  while(gap > 0)  begin  for(var i = gap, size)  begin  var temp: int = list[i];  var j: int = i;  while(j >= gap and list[j-gap] > temp)  begin  list[j] = list[j-gap];  j = j - gap;  end  list[j] = temp;  end  gap = Int(gap / 2);  end  **return** list;  end  def init(var args: string[]): **int**  begin  var size: int;  input(size);  var list: int[];  for(var i = 0, size, 1)  begin  input(list[i]);  end  for(var i = 0, size, 1)  begin  print(list[i]);  end  list = shellsort(list, size);  for(var i = 0, size, 1)  begin  print(list[i]);  end  **return** 0;  end |
| --- |

### 6.3 Fibonacci

| def fibonacci(var lim: int): **void**  begin  var result: int[] = [0, 1];  var i: int = 1;  while(result[i-1] + result[i] < lim)  begin  i = i + 1;  result[i] = result[i-2] + result[i-1];  end  var l: int = 0;  var step: int = 1;  for(l, i, step)  begin  print("%d ,", result[l]);  end  end  def init(var args: string[]): **int**  begin  var lim: int;  input(lim);  fibonacci(lim);  **return** 0;  end |
| --- |

# REFERÊNCIAS

LOOPS/FOR. **Rosetta Code**, [S. l.], p. 1, 31 out. 2021. Disponível em: https://rosettacode.org/wiki/Loops/For. Acesso em: 1 out. 2021.

CONDITIONAL structures. **Rosetta Code**, [S. l.], p. 1, 26 set. 2021. Disponível em: https://rosettacode.org/wiki/Conditional\_structures. Acesso em: 1 out. 2021.