UNIVERSIDADE FEDERAL DE ALAGOAS

INSTITUTO DE COMPUTAÇÃO

COMPILADORES 2021.1

# Especificação da Linguagem JORG

Gabriel Luiz Leite Souza

João Victor Falcão Santos Lima

Rodrigo Santos da Silva

Maceió

2021

# Introdução

A linguagem de programação “JORG” é voltada ao desenvolvimento de programas e algoritmos simples, com valor para a aprendizagem de programação ao possibilitar o uso de conceitos como operações com tipos primitivos de dados, arranjos unários, e estruturas de controle. O nome JORG vem dos nomes dos criadores (JOão, Rodrigo, Gabriel).

# Estrutura Geral do Programa

## Escopo

A linguagem utiliza escopo estático. Cada subprograma tem seu próprio escopo local definido por blocos de instruções, os quais são constituídos por um conjunto de instruções delimitadas por chaves (**{** … **}**), e cada instrução é terminada por um ponto e vírgula (**;**). Não são permitidos subprogramas aninhados.

No caso de instruções que fazem a abertura de um bloco em sua declaração (e.g. comandos **if**, **while**), é permitido o aninhamento destes comandos, mas o escopo do bloco aberto nesse caso é compartilhado com o escopo do subprograma respectivo. Logo, as declarações feitas dentro dele ficam visíveis para instruções e blocos seguintes, bem como as declarações feitas em blocos mais externos ou anteriores também são acessíveis, respeitando os limites do subprograma.

Além disso, qualquer variável ou constante nomeada declarada fora de um escopo local (logo fora de um subprograma) é considerada global.

## Ponto de Início

O Ponto de Início de um programa é sempre marcado pela função **main()**, que não exige retorno específico (por isso o tipo **void**). Exemplo:

```
function void main() {
    . . .
}
```

## Definições de Procedimentos e Funções

As funções e procedimentos devem ser declarados fora da função main. Como citado anteriormente, não é permitido o aninhamento de subprogramas. Além disso, para um subprograma ser chamado com sucesso, é preciso já ter declarado ele em um ponto antes da linha onde ocorre a chamada. Assim, os procedimentos e funções devem ser declarados antes do ponto de início do programa.

### Definições de Variáveis Globais e Locais

Variáveis declaradas fora do escopo de um subprograma (função ou procedimento) são consideradas globais e ficam acessíveis para os escopos declarados posteriormente. Já as variáveis definidas dentro de um escopo são locais, visíveis apenas por ele.

### Comentários

Os comentários são feitos apenas por linha. O caractere '#' marca o início de um comentário, e todo texto escrito após a ocorrência desse caractere até o final da linha é então ignorado pelos analisadores. Exemplo:

```
# Este é um comentário
```

## Nomes

### Sensibilidade à Caixa

A linguagem é sensível à caixa, logo "id" e "ID" são considerados identificadores diferentes. Além disso, todas as palavras reservadas da linguagem são escritas em caixa baixa.

### Limite de Tamanho

A linguagem JORG não limita o tamanho dos nomes.

### Formato (Expressão Regular)

Os nomes utilizados devem sempre seguir a expressão regular a seguir, `'([a-zA-Z])([a-zA-Z]|[0-9]|_)*'`. Isto significa que é necessário iniciar nomes com uma letra, e em sequência são permitidas múltiplas letras, dígitos e underlines, mas não aceitando qualquer outro símbolo.

## Tipos e Estruturas de Dados

### Forma de Declaração

As declarações seguem o padrão <**tipo**> **nome**, onde o termo tipo consiste em uma palavra reservada correspondente ao tipo de dado que será atribuído à variável. Mais de uma variável pode ser declarada na mesma instrução, e para isso basta utilizar uma vírgula para separar seus nomes. Atribuições podem ser realizadas no momento da declaração, ou após ela. Exemplos:

```
int a;
int a = 5;
int a, b, c;
int a = 10, b = 20;
```

## Tipos de Dados Primitivos

### Inteiro

Armazena valores numéricos inteiros com 32 bits. Declarado através da palavra reservada "**int**". As constantes literais são compostas por um ou mais dígitos, de acordo com a expressão regular `'([0-9])+'`. Com o tipo inteiro podemos realizar operações aritméticas, relacionais e de atribuição. Exemplo:

```
int a = 5;
```

### Ponto Flutuante

Armazena valores numéricos de ponto flutuante com 64 bits. Declarado através da palavra reservada "**float**". As constantes literais são compostas por um ou mais dígitos, seguidos de um ponto, e então novamente um ou mais dígitos, sendo esta a expressão regular `'([0-9])+(\.)([0-9])+'`. Com o tipo ponto flutuante podemos realizar operações aritméticas, relacionais e de atribuição. Exemplo:

```
float y = 5.5;
```

### Caractere

Armazena caracteres, através de códigos de 8 bits referentes à tabela ASCII. Declarado através da palavra reservada "**char**". Constantes literais do tipo caractere são definidas por apenas uma letra, dígito ou símbolo entre apóstrofes, de acordo com a expressão regular `'(\')(.)(\')'` (qualquer byte de caractere, desde que entre apóstrofes). Com o tipo caractere podemos realizar operações relacionais e de atribuição. Exemplo:

```
char letra_a = 'a';
```

### Booleano

Armazena valores booleanos, declarado através da palavra reservada "**bool**" e admite apenas dois valores, sendo eles **true** ou **false**, ambos palavras reservadas. Desse modo, as constantes literais também são esses valores, logo temos a expressão regular `'("true"|"false")'`. Com o tipo booleano podemos realizar operações relacionais, lógicas e de atribuição. Exemplo:

```
bool verdadeiro = true;
```

## Cadeias de Caracteres

Cadeias de caracteres são indicadas pela palavra reservada **string**. As constantes literais são formadas por uma sequência de zero ou mais caracteres (sem limite definido), todos entre

aspas, de acordo com a expressão ‘(“)(.)*(”)’. Com as cadeias de caracteres, podemos realizar operações relacionais, de concatenação e de atribuição. Exemplo:

```
string cadeia = “caracteres”;
```

A operação de concatenação é realizada através do operador ‘:’ (dois pontos), que combina o conteúdo de duas ou mais cadeias de caracteres em uma nova. Exemplo:

```
string cadeia_concatenada = “parte 1 ” : “parte 2”;

# resultado = “parte 1 parte 2”
```

## Arranjos

A linguagem suporta arranjos unidimensionais, que devem ser formados por elementos de um mesmo tipo primitivo. A declaração é semelhante à de uma variável tradicional do tipo, porém acrescida de colchetes (caracteres “[“ e “]”) ao lado do identificador, com um valor inteiro (constante ou não) entre eles indicando quantos elementos serão armazenados. Com arranjos podemos fazer apenas a operação de atribuição. Exemplo:

```
int array[10];
```

A atribuição pode ser feita no momento da declaração, onde deverá ser fornecida uma lista com os valores que preencherão o arranjo. A lista deve estar entre colchetes e com os valores separados por vírgula. Exemplo:

```
int arranjo[5] = [10, 20, 30, 40, 50];
```

Caso sejam fornecidos menos elementos, as posições restantes receberão o valor padrão do tipo respectivo ao arranjo. Caso sejam fornecidos mais elementos do que a capacidade do arranjo, será indicado um erro.

O armazenamento em memória é feito de modo estático, por isso é ressaltada a obrigatoriedade da indicação de seu tamanho no momento da declaração. Além disso, os elementos são indexados de 0 a *tamanho* - *1*. Assim, a referência/acesso a um elemento é feito pelo identificador do array acrescido de colchetes e um valor inteiro correspondente a um índice entre 0 e *tamanho* - *1*. Caso o valor seja negativo ou maior que *tamanho* - *1*, será causado erro. Exemplo:

```
arranjo[0] = arranjo[1] + arranjo[2];
```

Para verificar o tamanho de um arranjo, é utilizado o operador “**size**”, seguido do identificador do arranjo. Exemplo:

```
int tam = size arranjo;
```

### Equivalência de Tipos

A linguagem JORG possibilita a coerção implícita do tipo ponto flutuante para o tipo inteiro, de modo que em operações envolvendo valores desses dois tipos, será considerada apenas a parte inteira dos valores de ponto flutuante. O inverso não ocorre.

Também é possível a conversão de valores de qualquer tipo para cadeia de caracteres, em operações de concatenação. No caso de valores numéricos, seu valor literal é transformado em cadeia, no caso dos caracteres eles são transformados em cadeia de tamanho um e os booleanos viram uma cadeia correspondente ao valor literal (true/false).

Fora isso, não há suporte para outros tipos de coerção ou coerção explícita (cast).

### Constantes com Nome

Para a definição de constantes nomeadas, basta adicionar a palavra reservada "**const**" antes da declaração tradicional com **<tipo> nome**. Assim, a atribuição de um valor para essa variável só pode ocorrer uma única vez. Exemplo:

```
const int constante_nomeada = 10;
```

## Atribuição e Expressões

### Atribuição

Atribuições são realizadas com uso do operador '='. O lado que "recebe" a atribuição é o esquerdo (consistindo em identificadores de variáveis), e o lado do valor atribuído é o direito (outras variáveis, constantes literais ou nomeadas, chamadas de função). Exemplo:

```
variavel_inteira = 10;

cadeia = "caracteres";
```

### Operadores

De acordo com os operadores e seus operandos, as expressões da linguagem performarão ações específicas. Os operandos podem ser identificadores de variáveis, constantes literais, ou chamadas de função que retornam algum valor. Já os operadores englobam as categorias de operadores aritméticos, relacionais, lógicos e de concatenação, e serão discutidos nas seções a seguir.

#### Aritméticos

Os operadores aritméticos suportados são: adição/unário positivo (+), subtração/unário negativo (-), multiplicação (*), divisão (/), exponenciação (^) e resto (%), e atuam sobre valores numéricos (inteiros ou ponto flutuante).

## Relacionais

Os operadores relacionais suportados são: igualdade (==), desigualdade (!=), maior que (>), menor que (<), maior ou igual (>=), e menor ou igual (<=). Todos eles atuam sobre valores numéricos, caracteres e cadeias de caracteres. Para valores booleanos, apenas os operadores de igualdade e desigualdade são permitidos.

## Lógicos

Os operadores lógicos suportados são conjunção (&), disjunção (|) e negação unária (!). Atuam sobre valores booleanos.

## Concatenação

O operador de concatenação é ‘:’, e  atua sobre cadeias de caracteres, bem como os tipos primitivos, através da coerção implícita já explicada.

## Precedência e Associatividade

A tabela a seguir estabelece a associatividade dos operadores nas expressões, e está listada em ordem crescente de precedência:

| Operadores | Associatividade |
|---|---|
| ‘=’ | Não associativo |
| ‘:’ | Esquerda para Direita |
| ‘\|’ | Esquerda para Direita |
| ‘&’ | Esquerda para Direita |
| ‘!’ | Direita para Esquerda |
| ‘==’ ‘!=’ ‘>’ ‘>=’ ‘<’ ‘<=’ | Esquerda para Direita |
| ‘+’ ‘-’ | Esquerda para Direita |
| ‘*’ ‘/’ | Esquerda para Direita |
| ‘^’ ‘%’ | Direita para Esquerda |
| ‘+’ ‘-’ (unários) | Direita para Esquerda |
| ‘size’ | Não associativo |

Cabe notar que são permitidas expressões entre parênteses, alterando a precedência para resolver primeiro o conteúdo dos parênteses.

### Avaliação em curto-circuito

A avaliação em curto-circuito é empregada apenas para expressões lógicas. Assim, em casos como `bool result = (a >= 0) & (b >= 0)`, se tivermos a < 0, o restante da expressão não precisa ser avaliado, pois já sabe-se que o resultado será falso.

# Sintaxe e Exemplo de Estruturas de Controle

## Comandos de Seleção

A linguagem fornece comandos de seleção através das instruções "**if**" e "**else**", cada uma com um bloco de instruções associado, assim podemos ter seu uso por via única ou por via dupla. Para a instrução **if**, além do bloco de instruções, antes disso ela deve ser acompanhada de uma expressão lógica, ou constante/variável do tipo booleana que será avaliada. Caso a expressão tenha resultado verdadeiro, as instruções contidas no bloco do **if** serão executadas. Em via única, caso o resultado da expressão lógica do **if** seja falso, o bloco é ignorado e a execução do programa continua. Em via dupla, temos o comando **else**, sem expressão associada, e outro bloco de instruções, que sempre será executado no caso da expressão lógica do **if** ser falsa. Note que é necessário a presença do **if** anteriormente para que o **else** seja permitido. Exemplos:

```
if (expressao_logica) {
     # . . .
}

if (variavel_booleana) {
     # . . .
} else {
     # . . .
}
```

## Comandos de Iteração

### Controle por Contador

O comando de iteração por contador utilizado pela linguagem é o "**for**". Para seu uso, devem ser informados: uma variável contador, do tipo inteiro, declarada anteriormente ou dentro do for; e três valores inteiros, sejam eles constantes literais ou variáveis, correspondentes ao valor inicial do contador, ao valor final que deve ser atingido e o incremento que o contador receberá a cada iteração. O valor do incremento pode ser omitido, nesse caso é considerado o valor 1. Na primeira execução, o valor inicial é atribuído ao contador, e ao final de cada execução do bloco de instruções o contador é incrementado de acordo com o valor definido no **for**, até que o contador atinja (ou ultrapasse) o valor final.

```
for(int i, 0, 10, 1) {
     # ...
}


for(int j, 0, 10) {
     # ...
}
```

### Controle Lógico

O comando de iteração por controle lógico da linguagem é o "**while**". Ele deve ser acompanhado de uma expressão lógica (ou constante/variável booleana) e terá as instruções de seu bloco executadas enquanto o resultado desta expressão for verdadeiro. A verificação é por pré-teste, logo ocorre antes da entrada no bloco, e se na primeira execução do while a condição já falhar, o código contido no bloco não será executado. Exemplo:

```
while(expressao_logica) {
     # ...
}
```

Em ambos os tipos de comando de iteração, o comando "**break**" pode ser utilizado para encerrar o laço imediatamente.

## Desvios Incondicionais

Não há suporte para desvios incondicionais.

# Subprogramas

## Procedimentos e Funções

Funções são definidas com a palavra reservada **function**, seguida do **tipo** obrigatório de retorno (*int*, *string*, etc), o **identificador** único que nomeia a função, a lista de parâmetros entre parênteses, e finalmente o seu bloco de escopo; a última instrução do bloco deve ser o retorno de um valor de mesmo tipo que o indicado na declaração da função. Exemplo:

```
function tipoRetorno idFuncao(parametros) {
     . . .
     return valor;
}
```

Já os procedimentos são definidos como funções sem valor retornado. Desse modo, a declaração é a mesma, porém utilizando **void** no lugar do tipo de retorno. Exemplo:

```
function void idProcedimento(parametros) {
     . . .
}
```

Cabe notar que em procedimentos é permitido o uso do comando '**return;**' mesmo que sem um valor a ser retornado, neste caso apenas encerrando a execução do subprograma.

### Passagem de Parâmetros

A passagem de parâmetros de tipos primitivos e cadeias de caracteres é feita por valor, tornando-os variáveis locais no escopo do subprograma. No caso de arranjos, é feita por referência, alterando os dados externamente ao escopo do subprograma. Em ambos os casos, a passagem de parâmetros é feita de maneira posicional, por uma lista dos identificadores ou constantes, informada entre parênteses e com itens separados por vírgula, ao lado do identificador da função. Evidentemente, os valores devem ser passados de acordo com a ordem definida na declaração da função e seus tipos devem sempre corresponder. Exemplo:

```
funcao_exemplo(a, b, 10, true);
```

### Retorno de Resultados

O resultado retornado deve ser do mesmo tipo (ou de algum que permita coerção) que o indicado na declaração da função. O comando de retorno é indicado pela palavra reservada "**return**". Em casos de procedimentos, onde não há retorno de resultados, o comando de retorno apenas encerra a execução do procedimento onde foi declarado.

### Subprogramas como Parâmetros

Não é permitida a passagem de subprogramas como parâmetro.

## Entrada e Saída

### Comandos write e writeln

Os comandos **write** e **writeln** são os comandos de saída da linguagem JORG. Seu funcionamento é semelhante, aceitando uma lista de cadeia de caracteres (que pode ser uma constante literal, uma variável, o resultado da concatenação de várias cadeias, ou variáveis de outros tipos passando por coerção) como parâmetro e imprimindo cada uma delas na tela, em sequência. A diferença entre eles é que o **write** imprime todas as cadeias numa mesma linha, enquanto o **writeln** imprime uma quebra de linha após cada cadeia. Caso uma cadeia contenha o caractere '\n', será realizada uma quebra de linha, independentemente do comando ou se chegou ao fim da cadeia ou não. É necessário pelo menos uma cadeia de

caracteres para chamar o comando, logo para imprimir “nada”, é necessário fornecer uma cadeia vazia como parâmetro (“”) . Exemplo:

```
write(“hello\nworld”);
writeln(“hello”, 10, “world ” : 2);
#saída:
#hello
#worldhello
#10
#world 2
```

## Comando input

A entrada de dados é realizada por meio do comando **input**. O comando recebe uma lista de identificadores de variáveis, que devem estar previamente declaradas, e atribuirá os valores recebidos na entrada às variáveis na ordem em que foram listadas. Não é necessário indicar o tipo da variável, mas caso o valor fornecido na entrada não seja compatível com o da variável, poderão ocorrer erros ou inconsistências.

```
int a, b;
input(a, b);
```