Sumário

Lista de Tabelas

1. Estrutura básica de um algoritmo

Todo algoritmo possui um nome, um início e um fim. O nome do algoritmo deve ser delimitado entre aspas duplas depois do comando Algoritmo, como podemos visualizar na tabela 1. Um algoritmo pode, também, possuir variáveis, como veremos na seção **2** deste livro. Dessa forma, a sintaxe geral (estrutura) de um algoritmo em VisualG pode ser visualizada na tabela 1.

Linha	Código
1	**Algoritmo** "<nome_algoritmo>"
2	**Var** <lista_de_variáveis>
3	**Inicio**
4	<sequência_de_comandos>
5	**Fimalgoritmo**

Tabela 1 - Sintaxe geral de um algoritmo em VisualG

Como você pode notar na tabela 1, acima, existem palavras que são comandos, ou seja, durante a execução do algoritmo, quando o computador encontra essas palavras, ele executa uma instrução específica (semântica específica). Esses comandos são chamados de palavras reservadas. Na tabela acima, encontramos as seguintes palavras reservadas:

- **Algoritmo** – sinaliza que estamos criando um novo algoritmo;
- **Var** – sinaliza que queremos criar variáveis (ver seção **2**);
- **Inicio** – delimita o início do corpo principal do algoritmo, em que todas as instruções lógicas devem ser inseridas para que o algoritmo faça sentido e resolva um problema;
- **Fimalgoritmo** – delimita o fim do corpo principal do algoritmo.

Além das palavras reservadas, a tabela 1 mostra alguns campos delimitados entre parênteses angulares. Esses campos devem ser substituídos pelo desenvolvedor do algoritmo durante o processo de projeto, criação e escrita do código-fonte do algoritmo:

- <nome_algoritmo> – esse campo deve ser substituído pelo nome do algoritmo. Esse nome deve seguir as mesmas regras de nomeação de identificadores, como mostra a seção **2**;
- <lista_de_variáveis> – esse campo deve ser substituído pela lista de variáveis e seus respectivos identificadores e tipos (ver seção **2**);
- <sequência_de_comandos> – esse campo deve ser substituído pela sequência de instruções que fará o computador executar o algoritmo seguindo a lógica que leva à solução de um problema algorítmico.

Para criar seu primeiro algoritmo e entender melhor como utilizar as palavras reservadas e os campos da tabela 1, anterior, observe a tabela 2:

Linha	Código
1	**Algoritmo** "meuAlgoritmo"
2	**Var**
3	//não há variáveis
4	**Inicio**
5	escreva("Olá, mundo")
6	**Fimalgoritmo**

Tabela 2 - Exemplo de algoritmo

Observe que na linha 1 da tabela 2, o campo `<nome algoritmo>` foi substituído por `"meuAlgoritmo"`. Já na linha 3, o campo o campo `<lista de variáveis>` foi substituído pelo texto `//não há variáveis`. Aqui é importante ressaltar que os caracteres `//` denotam que essa linha será utilizada apenas como **comentário**, ao invés de comandos executáveis pelo computador. Nesse caso, o comentário feito pelo desenvolvedor teve o intuito de deixar documentado, no código, o fato de não haver variáveis declaradas (veremos sobre variáveis na seção **2**). Por fim, na linha 5, o campo `<sequência de comandos>` foi substituído pela instrução `escreva("Olá, mundo")` (ver seção **1.1**).

1.1. Operações de entrada e saída de dados para se comunicar com o usuário

Saída de dados: quando queremos mostrar informações de texto na tela, para o usuário, utilizamos basicamente duas funções:

- `escreva(<mensagem>)` – o campo `<mensagem>` deve ser substituído por textos literais, delimitados entre aspas duplas, e também pode-se imprimir o conteúdo de variáveis;
- `escreval(<mensagem>)` – a função `escreval()` é muito similar à função `escreva()`, com a diferença de que a função `escreval()` sempre quebra uma linha para que a as próximas mensagens de texto sejam impressas na linha de baixo, na tela do usuário.

Entrada de dados: por outro lado, quando precisamos que o usuário insira informações durante a execução do algoritmo, podemos utilizar a função `leia()`:

- `leia(<variáveis>)` – essa função permite que o usuário insira dados via teclado e, a cada *enter* teclado pelo usuário, um novo dado é inserido dentro da variável delimitada pelo comando `<variáveis>`. Através dessa função, é possível que o usuário preencha o conteúdo de uma única variável, ou várias delas, dependendo apenas da lógica do algoritmo.

Para entender melhor como utilizar os comandos de entrada e saída, observe a tabela 3:

Linha	Código
1	`Algoritmo "meuAlgoritmo"`
2	`    Var`
3	`    num: inteiro //(ver seção 2)`
4	`Inicio`
5	`    escreval("Digite um número:") //saída de dados`
6	`    leia(num)                     //entrada de dados`
7	`Fimalgoritmo`

Tabela 3 – Exemplo de entrada e saída

Durante a execução do algoritmo da tabela 3, quando o computador chegar à linha 5, aparecerá, para o usuário, a mensagem "Digite um número:" na tela, e então será pulada uma linha. Em seguida, na execução da linha 6, o computador irá aguardar até que o usuário digite um número inteiro e pressione *enter*. Após pressionar *enter*, o mesmo número digitado pelo usuário estará guardado na variável chamada `num`.

2. Variáveis

Variáveis são espaços em memória para que o algoritmo possa guardar dados utilizados durante sua execução. Toda variável deve ser declarada logo após o comando Var. Além disso, toda variável deve possuir um **identificador** próprio e um **tipo** de dados, como você pode observar na sintaxe abaixo:

Linha	Código
1	`Var <identificador>: <tipo>`

Tabela 4 – Sintaxe de declaração de uma variável em VisualG

Na tabela 4 temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<identificador>` – é o nome de uma variável. Esse nome geralmente começa com uma letra ou *underline* (*underscore*); nos caracteres intermediários, é possível utilizar números e *underlines*; o nome não pode começar com um número; o nome não admite espaços, acentos e nem caracteres especiais;
- `<tipo>` – existem basicamente quatro tipos de dados:
  - `inteiro`: dados numéricos sem casas decimais;
  - `real`: dados numéricos com casas decimais;
  - `logico`: dados booleanos (valem `verdadeiro` ou `falso`);
  - `caractere`: texto.

Para entender melhor como declarar variáveis, observe a tabela 5:

Linha	Código
1	`Algoritmo "meuAlgoritmo"`
2	`Var`
3	`    dado: real`
4	`    nome, sobrenome, endereco: caractere`
5	`    flag: logico`
6	`Inicio`
7	`    escreva("Olá, mundo")`
8	`Fimalgoritmo`

Tabela 5 – Exemplo de declaração de variáveis

Observando a linha 3, da tabela 5, podemos perceber que foi declarada a variável `dado`, do tipo `real`. Na linha 4, foram declaradas três variáveis distintas em uma única linha, todas do tipo `caractere`. Ou seja, da linha 2 à linha 5, foi possível declarar um total de cinco variáveis distintas, cada uma com seu próprio identificador e tipo.

2.1. Constantes

Constantes são espaços em memória que guardam um dado que não se alterará durante a execução do algoritmo. Para alterar o conteúdo de uma constante, sempre seremos obrigados a alterar o próprio código-fonte. As constantes devem ser declaradas após a palavra reservada `Const`, imediatamente antes da declaração das variáveis, como podemos ver na sintaxe a seguir:

Linha	Código
1	`Const <identificador> = <conteúdo>`

Tabela 6 – Sintaxe de declaração de uma constante em VisualG

Na tabela 6 temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<identificador>` – é o nome da constante. A escolha de um identificador para uma constante deve seguir as mesmas regras de nomeação de variáveis, vistas anteriormente;
- `<conteúdo>` – é o valor que será atribuído à constante, podendo ser um literal `inteiro`, `real`, `caractere` ou `logico`.

2.2. Atribuição

O operador de atribuição permite que o algoritmo modifique o conteúdo das variáveis em tempo de execução. A sintaxe para se realizar uma atribuição pode ser vista a seguir:

Linha	Código
1	`<variável> <- <dado>`

Tabela 7 – Sintaxe de atribuição de um valor a uma variável em VisualG

Na tabela 7, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<variável>` – nome (identificador) da variável que terá seu conteúdo alterado;
- `<dado>` – conteúdo a ser atribuído, podendo ser um literal, uma constante, uma variável ou o resultado de toda uma expressão.

Para entender melhor como realizar atribuição a variáveis, bem como o uso de constantes, observe a tabela 8:

Linha	Código
1	`Algoritmo "somar"`
2	`Const`
3	`   X = 0.0`
4	`Var numeroA, numeroB: inteiro`
5	`    resultado: real`
6	`    msg: caractere`
7	`Inicio`
8	`   leia(numeroA)`
9	`   leia(numeroB)`
10	`   resultado <- numeroA + numeroB`
11	`   escreval("Soma: ", resultado)`
12	`   resultado <- X`
13	`   msg <- "Zerando: "`
14	`   escreval(msg, resultado)`
15	`Fimalgoritmo`

Tabela 8 – Exemplo de atribuição a variáveis

Na linha 10 da tabela 8, temos a variável `resultado` recebendo a atribuição do valor das variáveis `numeroA` e `numeroB` somadas, ou seja, a variável resultado recebe todo o conteúdo de uma **expressão aritmética** (nesta linha também observamos o operador de soma, "+", que

veremos na seção **3**). Nesse caso, o conteúdo da variável `resultado` dependerá dos valores associados às variáveis `numeroA` e `numeroB` pelo usuário na execução das linhas 8 e 9, respectivamente. Na linha 12 temos outro exemplo de atribuição no qual o conteúdo de `resultado` recebe o conteúdo da **constante** `X` que, nesse caso possui o valor **literal** 0. As linhas 2 e 3 são o local onde tal constante `X` foi definida. Na linha 13, temos o texto "Zerando: " sendo atribuído à variável `msg`, do tipo `caractere`.

3. Operadores

Podemos realizar diversas operações matemáticas em nossos algoritmos. Através de operadores matemáticos, podemos compor comandos capazes de calcular expressões aritméticas, relacionais e lógicas.

3.1. Operadores aritméticos

Os operadores aritméticos exigem operandos numéricos inteiros ou reais e podem resultar, também, em números inteiros ou reais. São eles:

Operação	Símbolo	Exemplo	Lê-se
Adição (soma)	+	A + B	A mais B
Subtração	-	A - B	A menos B
Multiplicação	*	A * B	A multiplicado por B
Divisão	/	A / B	divisão real de A por B
Divisão inteira	DIV	A DIV B	divisão inteira de A por B
Resto de divisão inteira	%	A % B	A resto de divisão por B
Potenciação (exponenciação)	^	A ^ B	A elevado a B

Tabela 9 – Operadores aritméticos do VisualG

3.2. Operadores relacionais

As operações relacionais exigem operandos numéricos inteiros ou reais e resultam em um valor booleano (`verdadeiro` ou `falso`). Normalmente, tais operadores são utilizados para compor condições lógicas para estruturas de decisão ou estruturas de repetição, que veremos nas seções 5 e 6, respectivamente. São eles:

Operação	Símbolo	Exemplo	Lê-se
Maior	>	A > B	A maior que B
Maior ou igual	>=	A >= B	A maior ou igual a B
Menor	<	A < B	A menor que B
Menor ou igual	<=	A <= B	A menor ou igual a B
Igual	=	A = B	A igual a B
Diferente	<>	A <> B	A diferente de B

Tabela 10 – Operadores relacionais do VisualG

Na tabela 10, considere que as variáveis `A` e `B` contêm um dado que é do tipo `inteiro` ou `real`.

3.3. Operadores lógicos

As operações lógicas exigem operandos booleanos e resultam em um valor booleano (`verdadeiro` ou `falso`). Normalmente, tais operadores são utilizados para compor condições lógicas para estruturas de decisão ou estruturas de repetição, que veremos nas seções 5 e 6, respectivamente. São eles:

Operação	Palavra reservada	Exemplo
Negação	`nao`	`nao A`
Conjunção	`e`	`A e B`
Disjunção	`ou`	`A ou B`
Disjunção exclusiva	`xou`	`A xou B`

Tabela 11 – Operadores lógicos do VisualG

Na tabela 11, considere que as variáveis `A` e `B` contém um dado que é do tipo `logico`.

4. Sub-rotinas

Algoritmos são frequentemente modularizados em funções, para que um problema complexo possa ser subdividido em subproblemas mais simples e, normalmente, recorrentes. Existem basicamente dois tipos de sub-rotinas em VisualG: as funções e os procedimentos.

4.1. Funções intrínsecas

Para facilitar a vida dos desenvolvedores de algoritmos, algumas funcionalidades encontram-se pré-programadas em funções que são nativas ao VisualG. Geralmente são funções para cálculos matemáticos ou manipulação de textos e cadeias de caracteres, como podemos ver nas tabelas 12 e 13, a seguir.

Função	O que a função faz
`abs(A)`	Calcula o valor absoluto de `A`
`arccos(A)`	Calcula o arco cosseno de `A`
`arcsen(A)`	Calcula o arco seno de `A`
`arctan(A)`	Calcula o arco tangente de `A`
`cos(A)`	Calcula o cosseno de `A`
`cotan(A)`	Calcula a cotangente de `A`
`exp(A, B)`	Calcula o resultado de `A` elevado à potência `B`
`grauprad(A)`	Converte `A` de graus para radianos
`int(A)`	Remove a parte decimal e retorna apenas parte inteira de um real `A`
`log(A)`	Calcula o logaritmo na base 10 de `A`
`logn(A)`	Calcula o logaritmo neperiano de `A`
`quad(A)`	Calcula o quadrado de `A`
`radpgrau(A)`	Converte `A` de radianos para graus
`raizq(A)`	Calcula a raiz quadrada de `A`
`rand()`	Gera um número aleatório real que pode estar entre 0 e o limite 1

`randi(A)`	Gera um número aleatório inteiro que pode estar entre 0 o limite `A`
`sen(A)`	Calcula o seno de `A`
`tan(A)`	Calcula a tangente de `A`

Tabela 12 – Funções matemáticas do VisualG

Essas funções podem ser invocadas de maneira similar às funções de entrada e saída de dados. Devemos levar em consideração que as variáveis `A` e `B` da tabela 12 podem ser substituídas tanto por variáveis, quanto por expressões ou até mesmo por valores literais.
Existem ainda funções para manipulação de variáveis do tipo caractere. Ou seja, caso o desenvolvedor de algoritmos queira modificar e operar sobre texto, poderá utilizar as seguintes funções:

Função	O que a função faz
`asc(T)`	Calcula o valor absoluto de `A`
`carac(I)`	Calcula o arco cosseno de `A`
`caracpnum(T)`	Calcula o arco seno de `A`
`compr(T)`	Calcula o arco tangente de `A`
`copia(T, I, N)`	Calcula o cosseno de `A`
`maiusc(T)`	Calcula a cotangente de `A`
`minusc(T)`	Calcula o resultado de `A` elevado à potência `B`
`numpcarac(N)`	Converte `A` de graus para radianos
`pos(C, T)`	Remove a parte decimal e retorna apenas parte inteira de um real `A`

Tabela 13 – Funções de manipulação de caracteres do VisualG

Nas funções da tabela 13, considerar que `T` e `C` são variáveis do tipo `caractere`, `I` é do tipo `inteiro` e `N` pode ser `inteiro` ou `real`.

4.2. Criando novas funções

Caso você identifique a necessidade de criar sua própria função. Geralmente, declara-se uma sub-rotina antes do corpo principal do algoritmo, de acordo com a seguinte sintaxe:

Linha	Código
1	`Funcao <nome>([<parâmetros>]):<tipo_retorno>`
2	`  Var <variáveis-locais>`
3	`Inicio`
4	`  <bloco_de_comandos>`
5	`  retorne <valor_retorno>`
6	`Fimfuncao`

Tabela 14 – Sintaxe de declaração de uma nova função em VisualG

Na tabela 14, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<nome>` – nome (identificador) da função. Deve seguir as mesmas regras de nomeação e variáveis;
- `<parâmetros>` – variáveis de entrada da função. Uma função pode conter vários parâmetros, um único ou nenhum parâmetro. Tais variáveis de entrada devem ser declaradas uma após a outra, separadas por ponto-e-vírgula. Para cada uma delas, seu nome e seu tipo, de acordo com a seguinte sintaxe `<nome>: <tipo>`;

- `<tipo_retorno>` – é o tipo de retorno da função, que pode ser `real`, `inteiro`, `logico` ou `caractere`;
- `<bloco_de_comandos>` – aqui vai o passo a passo de comandos e instruções que compõem a sub-rotina;
- `<valor_retorno>` – aqui deve ser explicitado o dado que será retornado pela função. Esse dado pode ser um literal, uma constante, o conteúdo de uma variável ou o resultado de uma expressão. O `valor_retorno` deve ser do mesmo tipo que o `tipo_retorno`.

Para entender melhor como declarar, definir e invocar uma nova função, observe a tabela 15:

Linha	Código
1	`Algoritmo "novaFuncao"`
2	`Var`
3	`  param, retorno: real`
4	
5	`Funcao cubo(x: real):real`
6	`  Var resultado: real`
7	`Inicio`
8	`  resultado <- x*x*x`
9	`  retorne resultado`
10	`Fimfuncao`
11	`Inicio`
12	`   escreval("Insira o valor:")`
13	`   leia(param)`
14	`   retorno <- cubo(param)`
15	`   escreval("Resultado:", retorno)`
16	`Fimalgoritmo`

Tabela 15 – Exemplo de criação e chamada de uma função

Entre as linhas 5 e 10 da tabela 15, temos a declaração e definição da função `cubo()`. Como podemos ver, essa função precisa retornar um dado do tipo `real` (linha 5) e admite apenas um parâmetro `x`, também do tipo `real`. Internamente, na linha 6, é declarada uma variável local chamada resultado. Entre o `Inicio` e o `Fimfuncao` temos o corpo da função, no qual se calcula o conteúdo da variável resultado com base no valor de `x` (linha 8) e, na linha 9, a função retorna o conteúdo da variável `resultado` (repare como `resultado` é do tipo `real`, assim como a própria função). Uma vez declarada a função, ela pode ser invocada no corpo principal do algoritmo ou, eventualmente, poderia ser invocada dentro de qualquer outra função (inclusive ela própria). No nosso exemplo da tabela 15, na linha 14, invocamos a função `cubo()`, passando como argumento a variável `param` (que foi preenchida com um valor inserido pelo usuário, na linha 13). Após ser executada, ainda na linha 14, a função `cubo()` irá retornar seu resultado e atribuí-lo à variável `retorno`.

4.3. Criando procedimentos

Além das funções, existem os procedimentos. A diferença básica é que procedimentos não são obrigados a retornar dados de saída através do comando `retorne`. Geralmente, declara-se um procedimento antes do corpo principal do algoritmo, de acordo com a seguinte sintaxe:

Linha	Código
1	`Procedimento <nome>([<parâmetros>])`

```
2       Var <variáveis-locais>
3   Inicio
4       <bloco-de-comandos>
5   Fimprocedimento
```

Tabela 16 – Sintaxe de declaração de um novo procedimento em VisualG

Na tabela 16, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<nome>` – nome (identificador) do procedimento. Deve seguir as mesmas regras de nomeação e variáveis;
- `<parâmetros>` – variáveis de entrada (ou saída) do procedimento. Um procedimento pode conter vários parâmetros, um único ou nenhum parâmetro. Tais variáveis de entrada devem ser declaradas uma após a outra, separadas por ponto-e-vírgula. Para cada uma delas, seu nome e seu tipo, de acordo com a seguinte sintaxe `<nome>: <tipo>`;
- `<bloco_de_comandos>` – aqui vai o passo a passo de comandos e instruções que compõem a sub-rotina.

Observe como um procedimento não tem tipo, uma vez que ele não retorna dados através do comando `retorne`.

4.4. Passagem de parâmetros

Existem, basicamente, duas formas de se realizar a passagem de parâmetros a sub-rotinas: passagem de parâmetros por valor e passagem de parâmetros por referência.

4.4.1. por valor

A passagem de parâmetros por valor é aquela na qual apenas invocamos a sub-rotina e, em seus argumentos, informamos um valor que será armazenado em uma nova posição de memória. Ou seja, o parâmetro por valor guarda uma cópia do dado original. Caso essa cópia seja alterada, o dado original se mantém inalterado. A tabela 15 é um exemplo de declaração e utilização deste tipo de passagem de parâmetro em uma função.

4.4.2. por referência

Na passagem de parâmetros por referência o parâmetro definido no procedimento não guarda apenas uma cópia do dado original, mas sim manipula diretamente o dado original, como se ambos estivessem na mesma posição de memória. Em outras palavras, podemos dizer que a variável passada como argumento durante a invocação será alterada, caso o respectivo parâmetro seja alterado dentro da sub-rotina. Essa é a forma de se realizar o retorno de dados em procedimentos, através da passagem de parâmetros por referência. Para declarar parâmetros passados por referência, basta adicionar a palavra reservada `Var`, imediatamente antes da declaração do parâmetro, da seguinte forma `Var <parâmetro>: <tipo>`.

Para entender melhor como definir e declarar um procedimento e utilizar passagem de parâmetros por referência, observe a tabela 17:

Linha	Código
1	`Algoritmo "novoProc"`
2	`Var`
3	`  param, retorno: real`
4	`Procedimento cubo(x: real; Var res: real)`
5	`Inicio`
6	`  res <- x*x*x`
7	`Fimprocedimento`
8	`Inicio`
9	`   escreval("Insira o valor:")`
10	`   leia(param)`
11	`   cubo(param, retorno)`
12	`   escreval("Resultado:", retorno)`
13	`Fimalgoritmo`

Tabela 17 – Exemplo de procedimento com passagem de parâmetro por referência

Entre as linhas 4 e 7 da tabela 17, temos a declaração e definição do procedimento `cubo()`. Na linha 4 são definidos dois parâmetros: `x`, que é passado por valor e `res`, que é passado por referência (observe o comando `Var`). Entre o `Inicio` e o `Fimprocedimento` temos o corpo do procedimento, no qual se calcula o conteúdo da variável `res` com base no valor de `x` (linha 6). Ocorre que, pelo fato de `res` ser passado por referência, no momento da execução da linha 6, teremos que o conteúdo da variável `retorno` também será alterado, uma vez que `retorno` é o argumento que está relacionado ao parâmetro `res`.

5. Estruturas de decisão

Estruturas de decisão são comandos que definem blocos que só serão executados dependendo de uma condição.

5.1. Estrutura de decisão simples (`se-entao`)

A estrutura de decisão simples em VisualG é o `se-entao`. Com ela é possível delimitar um bloco de comandos que será executado somente se a expressão lógico-relacional estipulada for verdadeira. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`se (<expressão_lógica>) entao`
2	`      <sequência_de_comandos_1>`
3	`fimse`

Tabela 18 – Sintaxe de utilização do `se-entao` em VisualG

Na tabela 18, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<expressão_lógica>` – expressão lógico-relacional que deve ser avaliada antes de tentar executar o bloco de comandos;
- `<sequência_de_comandos_1>` – bloco de comandos que será executado caso a expressão lógico-relacional seja avaliada como verdadeira, em tempo de execução.

Para entender melhor como utilizar estruturas `se-entao`, em um contexto no qual o algoritmo deve decidir quem é o maior, entre dois números informados pelo usuário, observe a tabela 19:

Linha	Código
1	`Algoritmo "retornaMaior"`
2	`    Var num1, num2: inteiro`
3	`Inicio`
4	`    escreval("Informe dois números:")`
5	`    leia(num1)`
6	`    leia(num2)`
7	`    se (num1 > num2) entao`
8	`        escreva(num1)`
9	`    fimse`
10	`    se (num2 >= num1) entao`
11	`        escreva(num2)`
12	`    fimse`
13	`Fimalgoritmo`

Tabela 19 – Exemplo de utilização do comando `se-entao`

Na linha 7 da tabela 19 temos uma condição que, caso seja verdadeira, fará com que o comando da linha 8 seja executado para que, depois, o algoritmo siga seu fluxo normal a partir da linha 10. Caso a condição da linha 7 seja falsa, então a linha 8 é ignorada e o algoritmo continua seu fluxo a partir da linha 10.

5.2. Estrutura de decisão composta (`se-entao-senao`)

A estrutura de decisão composta em VisualG é o `se-entao`. Com ela é possível delimitar dois blocos de comandos distintos no qual apenas um deles será executado, dependendo da avaliação expressão lógico-relacional. Caso a expressão seja verdadeira, o bloco de comandos delimitado pelo `se-entao` será executado e o bloco posterior será ignorado. Caso contrário, se a expressão for falsa, então o bloco anterior será ignorado e será executado o bloco delimitado pelo `senao`. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`se (<expressão_lógica>) entao`
2	`    <sequência_de_comandos_1>`
3	`senao`
4	`    <sequência_de_comandos_2>`
5	`fimse`

Tabela 20 – Sintaxe de utilização do `se-entao-senao` em VisualG

Na tabela 20, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<expressão_lógica>` – expressão lógico-relacional que deve ser avaliada antes de tentar executar um dos blocos de comandos;
- `<sequência_de_comandos_1>` – bloco de comandos que será executado caso a expressão lógico-relacional seja avaliada como verdadeira, em tempo de execução;
- `<sequência_de_comandos_2>` – bloco de comandos que será executado caso a expressão lógico-relacional seja avaliada como falsa, em tempo de execução.

Para entender melhor como utilizar estruturas `se-entao-senao`, em um contexto no qual o algoritmo deve decidir quem é o maior, entre dois números informados pelo usuário, observe a tabela 21:

Linha	Código
1	`Algoritmo "retornaMaior"`
2	`    Var num1, num2: inteiro`
3	`Inicio`
4	`    escreval("Informe dois números:")`
5	`    leia(num1, num2)`
6	`    se (num1 > num2) entao`
7	`        escreval(num1)`
8	`    senao`
9	`        escreval(num2)`
10	`    fimse`
11	`Fimalgoritmo`

Tabela 21 – Exemplo de utilização da estrutura `se-entao-senao`

Na linha 6 da tabela 21 temos uma condição que, caso seja verdadeira, fará com que o comando da linha 7 seja executado, ignorando a linha 9, para que depois o algoritmo siga seu fluxo normal a partir da linha 10. Caso contrário, se a condição da linha 6 for falsa, então a linha 7 é ignorada, fazendo com que a linha 9 seja executada e, depois disso, o algoritmo continua seu fluxo a partir da linha 10.

5.3. Estrutura de decisão múltipla (`escolha-caso`)

A estrutura de decisão múltipla em VisualG, é o `escolha-caso`. Com ela é possível delimitar vários blocos de comandos no qual apenas um deles será executado, se a variável de controle for igual ao valor estipulado em um caso específico. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`escolha (<variável>)`
2	`    caso <valor_1>`
3	`        <lista_de_comandos_1>`
4	`    caso <valor_2>`
5	`        <lista_de_comandos_2>`
6	`    caso <valor_3>`
7	`        <lista_de_comandos_3>`
8	`    ⋮`
9	`    caso <valor_N>`
10	`        <lista_de_comandos_N>`
11	`    outrocaso`
12	`        <lista_de_comandos_padrão>`
13	`fimescolha`

Tabela 22 – Sintaxe de utilização do `escolha-caso` em VisualG

Na tabela 22, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<vairável>` – variável de controle cujo conteúdo é comparado com cada um dos valores relacionados a um caso respectivo;

- `<valor>` – valor comparado com o conteúdo da variável de controle para que se tente executar a `lista_de_comandos` relacionada ao respectivo `caso`;
- `<lista_de_comandos>` – bloco de comandos que será executado se a variável de controle for igual ao respectivo `valor` daquele `caso`;
- `<lista_de_comandos_padrão>` – lista de comandos que é executada se nenhum dos casos anteriores for atendido.

Para entender melhor como utilizar estruturas `escolha-caso`, em um contexto no qual o algoritmo deve receber um número inteiro e, em seguida, imprimir na tela o dia da semana correspondente àquele número, observe a tabela 23:

Linha	Código
1	`Algoritmo "diaDaSemana"`
2	`    Var num: inteiro`
3	`Inicio`
4	`    escreval("Informe um número de 1 a 7:")`
5	`    leia(num)`
6	`    escolha (num)`
7	`        caso 1`
8	`           escreval("Domingo")`
9	`        caso 2`
10	`           escreval("Segunda-feira")`
11	`        caso 3`
12	`           escreval("Terça-feira")`
13	`        caso 4`
14	`           escreval("Quarta-feira")`
15	`        caso 5`
16	`           escreval("Quinta-feira")`
17	`        caso 6`
18	`           escreval("Dia da maldade")`
19	`        caso 7`
20	`           escreval("Sábado")`
21	`        outrocaso`
22	`           escreval("Opção inválida")`
23	`    fimescolha`
24	`Fimalgoritmo`

Tabela 23 – Exemplo de utilização da estrutura `escolha-caso`

Na linha 6 da tabela 23 temos a definição de `num` como sendo a variável de controle da estrutura de decisão múltipla. Em cada uma das linhas de 7 a 2, temos um `caso` distinto. Por exemplo, se durante a execução da linha 5 o usuário inserir um valor igual a 7 para `num`, a estrutura de seleção irá comparar `num` com todos os valores elencados em cada um dos `caso`, até que finalmente o `caso 7` seja atendido e a mensagem "Sábado" (linha 20) seja impressa na tela. Se, eventualmente, o usuário preencher `num` com um valor que está fora do intervalo fechado de 1 a 7, então as linhas 21 e 22 serão executadas.

6. Estruturas de repetição

Estruturas de repetição são comandos utilizados para definir blocos que serão executados do início ao fim, repetidas vezes, com base em condições ou com base em contagem.

6.1. Estrutura **enquanto**

A estrutura de repetição `enquanto`, em VisualG, realiza a iteração de um bloco de comandos por várias vezes, contanto que a expressão lógico-relacional seja verdadeira. No momento em que a expressão se tornar falsa, o laço é interrompido e o bloco de comandos é ignorado. Sempre, antes de tentar iniciar uma nova repetição, a expressão lógico-relacional é avaliada novamente. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`enquanto (<expressão_lógica>) faca`
2	`    <sequência_de_comandos>`
3	`fimenquanto`

Tabela 24 – Sintaxe de utilização do `enquanto` em VisualG

Na tabela 24, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<expressão_lógica>` – expressão lógico-relacional que deve ser avaliada antes de tentar executar o bloco de comandos;
- `<sequência_de_comandos>` – bloco de comandos que será executado por uma vez após expressão lógico-relacional ser avaliada como verdadeira, em tempo de execução. Ao término da execução desse bloco de comandos, a `expressão_lógica` é avaliada novamente, na tentativa de se iterar mais uma vez essa própria `sequência_de_comandos`.

Para entender melhor como utilizar estruturas `enquanto`, em um contexto no qual o algoritmo deve garantir que o usuário insira um valor maior ou igual a zero, repetindo a solicitação de inserção de valor caso o usuário insista em inserir um valor inválido, observe a tabela 25:

Linha	Código
1	`Algoritmo "maiorQueZero"`
2	`    Var val: inteiro`
3	`Inicio`
4	`    val <- -1`
5	`    enquanto (val < 0) faca`
6	`        escreval("insira um valor:")`
7	`        leia(val)`
8	`    fimenquanto`
9	`    escreval("Valor validado:", val)`
10	`Fimalgoritmo`

Tabela 25 – Exemplo de utilização da estrutura `enquanto`

Na linha 4 da tabela 25, notamos que a variável `val` deve ser inicializada com um conteúdo que satisfaça a condição da linha 5, inicialmente, no intuito de que as linhas 6 e 7 sejam executadas por pelo menos uma vez. Na linha 7, o usuário é convidado a inserir um novo conteúdo a `val` e, após a execução dessa linha, o teste condicional da linha 5 é verificado novamente. Se o teste condicional da linha 5 resultar em verdadeiro, significa que o usuário não inseriu um valor válido, e o bloco de comandos é iterado mais uma vez. Caso contrário, a estrutura de repetição é encerrada e o algoritmo continua seu fluxo normal a partir da linha 9.

6.2. Estrutura **repita**

A estrutura de repetição `repita` em VisualG realiza a iteração de um bloco de comandos por várias vezes, contanto que a expressão lógico-relacional seja falsa. No momento em que a expressão se tornar verdadeira, o laço é interrompido e o bloco de comandos é ignorado. Sempre, depois de executar o bloco de comandos por uma vez, a expressão lógico-relacional é avaliada novamente, para só então tentar iterar o bloco de comandos novamente. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`repita`
2	`      <sequência_de_comandos>`
3	`ate(<expressão_lógica>)`

Tabela 26 – Sintaxe de utilização do `repita` em VisualG

Na tabela 26, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<sequência_de_comandos>` – bloco de comandos que é executado ao menos uma vez. Ao término da execução deste bloco, verifica-se a `expressão_lógica` e, caso seja verdadeira, a `sequência_de_comandos` é iterada mais uma vez;
- `<expressão_lógica>` – expressão lógico-relacional que deve ser avaliada depois de tentar se executar o bloco de comandos.

Para entender melhor como utilizar estruturas `repita`, em um contexto no qual o algoritmo deve garantir que o usuário insira um valor maior ou igual a zero, repetindo a solicitação de inserção de valor caso o usuário insista em inserir um valor inválido, observe a tabela 27:

Linha	Código
1	`Algoritmo "maiorQueZero"`
2	`     Var val: inteiro`
3	`Inicio`
4	`     repita`
5	`          escreval("insira um valor:")`
6	`          leia(val)`
7	`      ate(val >= 0)`
8	`     escreval("Valor validado:", val)`
9	`Fimalgoritmo`

Tabela 27 – Exemplo de utilização da estrutura `repita`

Inicialmente, nas linhas 5 e 6 da tabela 27, o usuário é, obrigatoriamente, solicitado a preencher o valor de `val` ao menos uma vez. Ao término da execução da linha 6, a expressão da linha 7 é avaliada. Caso a condição seja verdadeira, o bloco de comandos das linhas 5 e 6 será executado mais uma vez e então a expressão será reavaliada, caso contrário, o algoritmo segue seu fluxo normal, a partir da linha 8.

6.3. Estrutura **para**

A estrutura de repetição `para` em VisualG realiza a iteração de um bloco de comandos por várias vezes, iniciando uma contagem que parte de um valor inicial e vai até um valor final, incrementando ou decrementando a contagem, conforme a necessidade. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`para <v> de <i> ate <f> passo <p> faca`
2	`    <sequência de comandos>`
3	`fimpara`

Tabela 28 – Sintaxe de utilização do `para` em VisualG

Na tabela 28, acima temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<v>` – variável de controle que é utilizada para efetuar a contagem;
- `<i>` – valor inicial do qual a variável de controle deve partir;
- `<f>` – valor final que deve ser extrapolado pela variável, após uma ou mais iterações;
- `<p>` – valor de incremento ou decremento da variável de controle, para que ela possa ter seu conteúdo alterado, partindo de `i` e chegando até `f`.

Para entender melhor como utilizar estruturas `repita`, em um contexto no qual o algoritmo deve imprimir todos os números inteiros compreendidos no intervalo fechado de 1 a 10, pulando de linha a cada impressão, observe a tabela 29:

Linha	Código
1	`Algoritmo "contaDe1a10"`
2	`    Var cont: inteiro`
3	`Inicio`
4	`   para cont de 1 ate 10 faca`
5	`        escreval(cont)`
6	`   fimpara`
7	`Fimalgoritmo`

Tabela 29 – Exemplo de utilização da estrutura `para`

Na linha 4 da tabela 29, podemos observar que a variável de controle `cont` deve se iniciar em 1 e ser incrementada até atingir o limite que é igual a 10. Um detalhe importante é que, na linha 4, o comando passo foi omitido. Nesse caso, assume-se que o incremento é igual a 1.

7. Estruturas de dados homogêneas e heterogêneas

Para armazenar mais de um dado em uma variável de mesmo nome é preciso recorrer a estruturas de dados homogêneas e heterogêneas, a saber: vetores, matrizes, registro ou uma combinação entre eles.

7.1. Vetores

Vetores são como um edifício no qual há apenas um apartamento por andar. Para acessar cada andar, é necessário saber o número do andar. Feita essa analogia, em algoritmos, podemos definir um vetor como um conjunto de dados referenciados por um único nome. Vetores são controlados por um único índice que varia da primeira (índice igual a 1) à última (índice igual a N – onde N é o tamanho máximo do vetor) posição do vetor. Dessa forma, todos os elementos de um vetor ficam dispostos linearmente, de maneira unidimensional. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`<nome_do_vetor>: Vetor[1..<fim>] de <tipo>`

Tabela 30 – Sintaxe de declaração de um vetor unidimensional em VisualG

Na tabela 30, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<nome_do_vetor>` – identificador do vetor. Segue as mesmas regras de nomenclatura de variáveis;
- `<fim>` – tamanho do vetor;
- `<nome_do_vetor>` – tipo de dados do vetor.

Para entender melhor como utilizar vetores, em um contexto no qual o algoritmo deve solicitar ao usuário que preencha os dados de um vetor de 5 posições e depois, feito isso, o algoritmo deve imprimir todo o conteúdo do vetor, na tela, observe a tabela 31:

Linha	Código
1	`Algoritmo "vetorizando"`
2	`    Var vet: Vetor[1..5] de real`
3	`        i: inteiro`
4	`Inicio`
5	`    Para i de 1 ate 5 passo 1 faca`
6	`        escreval("Insira o dado da posição ", i, ":")`
7	`        leia(vet[i])`
8	`    Fimpara`
9	`    escreval("Conteúdo de vet:")`
10	`    Para i de 1 ate 5 passo 1 faca`
11	`        escreval(vet[i])`
12	`    Fimpara`
13	`Fimalgoritmo`

Tabela 31 – Exemplo de utilização de um vetor

Na linha 2 da tabela 31, observamos a declaração do vetor `vet`, de tamanho 5, do tipo `real`. Além disso, aproveitamos para criar uma variável auxiliar, `i`, que será utilizada para controlar o índice de acesso a cada uma das posições do vetor. Da linha 5 à linha 8 temos a estrutura de repetição que faz a leitura dos dados do vetor a partir das entradas do usuário (linha 7). Após ter preenchido todo o conteúdo do vetor, o algoritmo faz a impressão desse conteúdo na estrutura de repetição que vai da linha 10 à linha 12.

7.2. Matrizes

Matrizes são arranjos de dados multidimensionais. Como os vetores, é preciso acessar suas posições através de índices, todavia, no caso das matrizes, é preciso utilizar dois ou mais índices para isso. As matrizes mais comuns são as bidimensionais, nas quais temos um índice para linhas e outro índice para colunas. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`<nome_mat>: Vetor[<1..d1>,<1..d2>,...,<1..dN>] de <tipo>`

Tabela 32 – Sintaxe de declaração de uma matriz multidimensional em VisualG

Na tabela 32, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<nome_mat>` – identificador do vetor. Segue as mesmas regras de nomenclatura de variáveis;
- `<d>` – tamanho máximo de cada uma das dimensões da matriz;
- `<tipo>` – tipo de dados da matriz.

Para entender melhor como utilizar matrizes, em um contexto no qual o algoritmo deve solicitar ao usuário que preencha os dados de uma matriz de 3 linhas por 3 colunas e depois, feito isso, o algoritmo deve imprimir todo o conteúdo da matriz, na tela, observe a tabela 33:

Linha	Código
1	`Algoritmo "matricial"`
2	`     Var mat: Vetor[1..3, 1..3] de real`
3	`         i, j: inteiro`
4	`Inicio`
5	`   Para i de 1 ate 3 passo 1 faca`
6	`      Para j de 1 ate 3 passo 1 faca`
7	`        escreval("Insira o dado da linha ", i,", coluna ", j,":")`
8	`         leia(mat[i,j])`
9	`      Fimpara`
10	`   Fimpara`
11	`   escreval("Conteúdo de mat:")`
12	`   Para i de 1 ate 3 passo 1 faca`
13	`      Para j de 1 ate 3 passo 1 faca`
14	`        escreva(mat[i,j])`
15	`      Fimpara`
16	`      escreval()`
17	`   Fimpara`
18	`Fimalgoritmo`

Tabela 33 – Exemplo de utilização de uma matriz

Na linha 2 da tabela 33, observamos a declaração do vetor multidimensional (matriz) `mat`, de dimensões iguais a três linhas por três colunas, do tipo `real`. Além disso, aproveitamos para criar duas variáveis auxiliares, `i` e `j`, que serão utilizadas para controlar os índices de acesso a cada uma das posições da matriz. Da linha 5 à linha 10 temos as estruturas de repetição aninhadas que fazem a leitura dos dados da matriz a partir das entradas do usuário (linha 8). Após ter preenchido todo o conteúdo da matriz, o algoritmo faz a impressão desse conteúdo nas estruturas de repetição aninhadas que vão da linha 12 à linha 17.

7.3. Registros

Registros são conhecidos como estruturas de dados heterogêneas pois permitem que, em uma mesma variável, sejam armazenados conteúdos de tipos de dados distintos. Através de registros é possível abstrair o mundo real e criar novos tipos de dados que atendam melhor às necessidades dos desenvolvedores. Observe a sintaxe de utilização, a seguir:

Linha	Código
1	`Tipo`
2	`     <identificador> = registro`
3	`     <lista_dos_campos_e_seus_tipos>`
4	`Fimregistro`

Tabela 34 – Sintaxe de declaração de um registro em VisualG

Na tabela 34, acima, temos os seguintes campos a serem substituídos, no momento da construção do algoritmo:

- `<identificador>` – identificador do novo tipo de dados (registro). Segue as mesmas regras de nomenclatura de variáveis;
- `<lista_dos_campos_e_seus_tipos>` – relação de campos do registro. Cada campo deve possuir seu próprio identificador. Quanto ao tipo dos campos, podemos utilizar os tipos primitivos de dados (`inteiro`, `real`, `logico` e `caractere`).

Para entender melhor como utilizar registros, em um contexto no qual o algoritmo deve solicitar ao usuário que preencha os dados de cinco produtos que possuem, cada um deles, os seguintes campos: código, nome, quantidade e valor unitário, observe a tabela 35:

Linha	Código
1	`Algoritmo "exemploRegistros"`
2	`  Tipo`
3	`    produto = registro`
4	`    codigo: inteiro`
5	`    nome: caractere`
6	`    quantidade: inteiro`
7	`    valor: real`
8	`  Fimregistro`
9	`Var`
10	`  lista_prod: Vetor[1..5] de produto`
11	`  i: inteiro`
12	`Inicio`
13	`  para i de 1 ate 5 passo 1 faca`
14	`    escreva("Digite o código ", i, ":")`
15	`    leia(lista_prod[i].codigo)`
16	`    escreva("Insira o valor ", i, ":")`
17	`    leia(lista_prod[i].valor)`
18	`    escreva("Digite o nome ", i, ":")`
19	`    leia(lista_prod[i].nome)`
20	`    escreva("Insira a quantidade ", i, ":")`
21	`    leia(lista_prod[i].quantidade)`
22	`  fimpara`
23	`Fimalgoritmo`

Tabela 35 – Exemplo de utilização de um vetor de registros

Das linhas 2 a 8, na tabela 35, temos a definição do novo tipo de dados, um registro chamado `produto`. Nas linhas de 4 a 5 podemos identificar cada um dos quatro campos declarados, cada um com seu próprio identificador, e um tipo associado a ele. Na linha 10 declaramos `lista_prod`, que é um vetor de `produto`, no intuito de poder armazenar os dados de cinco produtos distintos. A estrutura de repetição que vai da linha 13 à linha 22 implementa a lógica de preenchimento de cada um dos campos de cada produto.